MASTER IN COMPUTER SCIENCE

PARFOR. An experimental parallel environment for fortran programmers

Noel, Roland

Award date:
1988

Awarding institution:
University of Namur

Facultés Universitaires Notre-Dame de la Paix.
Institut d'Informatique.
Namur, Belgique.

PARFOR

AN EXPERIMENTAL PARALLEL ENVIRONMENT FOR FORTRAN PROGRAMMERS

Mémoire présenté par
Roland NOEL
en vue de l'obtention du titre de
Licencié et Maître en Informatique.

Promoteur : Mr J. Ramaekers.

Année académique 1987-1988.

Résumé

Le document présent décrit les résultats d'une recherche dans le domaine de la programmation parallèle . Le sujet principal de la présentation est l'environnement PARFOR et les possibilités qu'il offre . Cet environnement est conçu dans le but de permettre la réécriture de programmes FORTRAN existants , de manière telle qu'ils puissent être exécutés avec un nombre quelconque de processus .

L'origine de la définition de PARFOR remonte à une pré-étude faite par une équipe de recherche de SIEMENS Munich . Dans le travail présent , plusieurs implémentations de cet environnement sont réalisées . Les outils fournis par PARFOR et la façon de les utiliser sont évalués . Pour ces réalisations , la machine MX500 SIEMENS , un multiprocesseur travaillant sous UNIX , est employée . Ensuite , quelques comparaisons sont faites avec un autre environnement parallèle existant - FORCE . Plusieurs tests sont envisagés , avec pour but l'évaluation des performances de PARFOR . La plupart des résultats de ces tests sont fournis sous forme graphique .

Summary

This document presents and describes the results of a research made in the area of parallel programming . The main subject is the presentation of the PARFOR environment and its actual possibilities . This environment is designed to allow the rewriting of existing FORTRAN applications so that they can take entire profit of a variable number of processes for their execution .

The starting point of the definition of PARFOR was made in a research team at SIEMENS Munich . In this work , Some implementations of this environment are tried ; the facilities provided and the ways they can be used are evaluated . For all this work , the MX500 SIEMENS machine , which is a full multiprocessor machine working under UNIX , is used . Then , some comparisons with a concurrent existing parallel environment - FORCE - are made . Many tests are envisaged in various situations for the evaluation of the performances of PARFOR , and most of the results are provided on graphics .

Acknowledgements

I would like to thank Mr FRIELINGHAUS , head of the D ST SP 3 department of SIEMENS AG Munich Perlach , for the possibility he provided me to have an interesting subject of research , and for the large number of facilities I got from his department to complete this work in the best conditions in Munich .

My gratitude also goes to Mr HARTINGER , Mr MÜNCHHAUSEN and Dr KAHL who supported my work during the whole time I was in Munich in 1986 and 1987 ; who gave me many opportunities to get relations with various people inside and outside the company at meetings , during discussions , and in the daily life ; and for the helpfull aids , advices , and encouragements they gave me .

In the same way , I would like to thank some collaborators : Heiner TITTELBACH , Henry GOFFIN , Matthias OLK , Helen STAPLETON , Daniel TORNO , Francois SCHMITZ , Francoise MIANE , and the other French and German people who also provided me some help .

Eventually , on the academic side , I must thank Mr RAMAEKERS , director of the Institute of informatics in Namur , for the facilities of contacts he gave me , with the responsible people of both SIEMENS Software Namur, and SIEMENS Munich ; and for the help in the preparation of this work .

Table of contents

Chapter 1

Introduction

Sequentiality was since a long time the only way to think in the growing world of computers and programs . Sequentiality is still at the present time the classic way of programming small and middle range computers , and also large computers .

But this situation is more and more changing . Many computers that are designed now are conceived with the basic idea of parallelism . Parallelism at various levels , from the hardware level to the highest software levels .

The object of the present work is the presentation of the design and the facilities of an experimental parallel environment for FORTRAN programmers . This environment is called PARFOR , and its target is to give to the FORTRAN programmer the possibilty to rewrite its sientific applications so that they can be executed by many instruction flows on a multiprocessor machine . The starting point of the definition of PARFOR was made by a research team at SIEMENS Munich in collaboration with the TU Munich .

The subject of this work is to test and evaluate the PARFOR environment within the UNIX operating system . Then we try to optimize its implementation whithin the available UNIX environment . Later , some comparisons are made with a concurrent in FORTRAN like parallel languages : The FORCE environment developped earlier in USA by Professor JORDAN and its team .

This work is structurated in the following way :

Chapter one is rather general and remembers some elementar theoretical concepts concerning parallelism , tools for parallel programming , some elements of theories for performances measurements , practical measures that we designed to take the measures in the parallel programs , and eventually , a review of the main general technics for parallel programming .

In a second chapter , the programmation in the PARFOR environment is explained , with all the problems that this involves , the various ways to program within PARFOR , the relations between the various concepts of PARFOR . Then , some comparisons are made with the possibilities provided by the FORCE environment .

The third chapter contains a description of the PARFOR and the FORCE implementations in the UNIX environment , and the modifications which were introduced in order to diminish the overheads when programming with PARFOR .

The next chapter describes some of the tests and the results we designed for the evaluation of the PARFOR environment . We describe also one test made in the FORCE environment , used to compare in a more direct way the facilities and the possibilities of PARFOR against FORCE .

Eventually , chapter five of this work contains a brief conclusion concerning this work , the conditions in wich it was done , the results obtained and the possible future of PARFOR .

Chapter 2

Elements of parallel programs

2.2 Parallelism

2.2.1 Introduction

Trying a classification of parallelism is a very sensible and delicate will . Parallelism in programs , as we conceive it in an abstract way , would consist in a possibility that allows programs or applications to be executed in a parallel way and at the same time . However , this abstract hope , at the present time , has found his real meaning only in a very restricted number of types of treatements , and in only a few types of applications . Actually , there exists not yet a general method of implementation of this kind of abstract parallelism , allowing a complete automatization , as we can imagine it . Of course , some specific technics try to face this religious wish . However , these technics , altogether , are not sufficient to form a basic general theory , but rather , they form only a set of technics , directly dependent of the type of application , of the type of the target machine which is employed , of the type of parallelism whished , and eventually , of many other factors on which we have presently no control or simply that we ignore .

But first , we should try to have a definition of a computer system . A computer system , in the large sense , is made of many hardware units , each of them beeing designed to perform some specific tasks . For this reason of specificity , it is not yet possible to solve general case of abstract parallelism that we mentioned earlier . In the reality , the parallelism is present as early as many of the units begin to work

independently , controlled by software . But this reality is still far from the abstract form of parallelism that we described above .

Some guesses of classification of parallelism can however be made . The better known is the Flynn's one , consisting of 3 axes . Another classification can be made on the conception that we have of parallelism , so it is a more abstract classification , and necessary more blurred . The third classification of the parallelism that we tried , is based on the concept of granularity of the sections of a program .

2.2.2 Classification of Flynn

The classification of the parallelism introduced by Flynn , is based on 3 main concepts .

The first considers the number of instructions and datas that the machine can treat simultaneously . Using this axe leads to a partition of 4 classes of systems , coded by SISD , MIMD , SIMD , MISD .

The second axe is a description of the interconnections between the memories and the processors . In this axe , the target of analyse is the topology of the system .

The third axe of the classification concerns the level of which an instruction flow can recover its operations . Flynn calls this the ' inertia factor ' , which can be described as the Pipe-line factor of the machine .

This classification is far away from perfect and is more and more desuet due to the diversification of the machines .

2.2.3 Partial classifications of the parallelism

Some partial rules can help for the classification of the parallelism , according to the perception that we have about it .

Parallelism according to his abstract perception :

Parallelism , as we consider it in an abstract way , is a target of research . Some people are trying to "resolve" it in the general case , and in this mind , are trying to improve technics to more general applications . So , the guess is to remain at the highest level of abstraction during the conception of parallel projects . That is , for example , the case in artificial intelligence .

Parallelism determined by a given architecture :

Computers are always in evolution to more complex hardwares and architectures . Their types are going to more and more diversifications , and the applications try to exploit more fully the new advantages provided by these improvements . That is a reason why new technics of parallelism are developped , following the multiple types of architectures available or , in development .

Parallelilsm determined by a type of application :

Another approach to the conception of the systems working with parallelism , is the conception of an architecture according to the structure of the problem to be treated later with the machine . A number of machines have seen their architecture copied on the structure of the most important types of problems for which they were designed . It is essentially the case in sciences , where some architectures are based on a structure which can easely and quickly solve the problems with matrix operations .

2.2.4 Classification of parallelism according to granularity

The granularity is an essential factor in the process of designing parallel applications . It defines the mean size of the grain of a parallel progam . This granularity is mainly dependent of the intrication of the code , and on the data dependencies in the application . The granularity of a program can widely vary from one application to another .

A coarse granularity :

A large granularity , is often present in applications in which the operating system itself manages many processes that are part of the global application . In such cases , the application uses calls to system procedures like FORK() and JOIN() , to be executed by parallel processes . Coordination middles are invoked to control the correctness of the running tasks . This type of parallelism is relatively simple and must be explicitely specified by the programmer , when designing the application .

A fine granularity :

A fine granularity is present in certain types of architectures . Particulary, the machines concerned are pipe-line machines or data-flow machines . The management of this kind of parallelism is transparent to the programmer . The conception of parallel applications is then simplifed . If it is transparent to the programmer , the programming language must care for particular formulations which can take profit of this parallelism .

A mean granularity :

This type of granularity is relatively frequent . It includes both characteristics of programs with large and fine granularities . Conventionnal languages allowing a certain amount of parallelism , dispose of special libraries containing functions to activate the parallel environment .

2.2.5 Distribution of the parallelism

Studies concerning the distribution , in a program , of the portion that can be parallelized and the part can not be parallelized have showed that in most of the applications , 99% of the code could be executed in parallel . But the main problem is that it is not always easy to redraw a parallel code from the sequential code . The reality shows often that the code is so intricated that it is not possible without large effort , to parallelize it . Often , the parallelizable sections are very fine so that in a given environment , the parallelism is not exploitable . It is for example the

case in loops , where assignements could be done in parallel . So the granularity factor is very important . In general , a coarse granularity is bound with a small number of independent tasks running in parallel .

2.2.6 The degree of parallelism

By the definition , the degree of parallelism is the number of tasks which can be executed in parallel , simultaneously . It is greatly dependent on the size of the problem to manage . But usually , large problems could provide large degrees of parallelism . In fact , more the problem is large , more the treatements it includes are repetitive , so that large degree of parallelism can easely be exploited .

The degree of parallelism
is highly coupled with the concept of granularity . If the tasks are very small (fine granularity) , the efficiency of the parallelism is very dependent of the overheads produced by the exploitation of this parallelism .

2.3 Introduction to the main basic concepts

2.3.1 Definitions

Multiprocessor : A multiprocessor is a computer or a computer system containing many processors similar or not , and a main global memory . The processors can work in parallel , share the common memory and the peripherals . By definition , a multiprocessor supports multiprocessing .

Parallel machine : A parallel machine is defined as a machine which , in one or another way , is conceived to treat problems of parallelism . We define as 'parallel treatement' a treatement that exploits a parallel environment .

Explicit parallelism : It is the possibility given to the programmer to execute in parallel more than one treatement depending or not , one on the other . The explicit parallelism is said of a program that is explicitely designed to be executed on a parallel machine . This concept is often bound to the concept of synchronization .

Implicit parallelism : This kind of parallelism groups all types of parallelisms which are not explicitely specified by the programmer .

Multiprogramming : This is the characteristic of an operating system to make reside in central memory many unrelated programs , and to make execute them in an imbricated way by the same central unit . The interrupt system of the machine can switch from one program to another .

Multitasking : This is the characteristic of an operating system to allow a job to be executed with more than one task . The tasks are executed in parallel , but not necessarily simultaneously. Multitasking implies the possibility of multitasking . On a multiprocessor , the tasks can be executed simultaneously .

Multiprocessing : This is the characteristic of an operating system to allow the execution of a program by a multiprocessor , so that many processors execute the same program at the same time . They share a common memory .

Process : A process is an instruction flow , a simple execution of a program which can be executed independently by the operating system .

Task : A task is a program unit which can be managed by a job . A task is typically a single instanciation of a subroutine , or a loop , which is executed simultaneously with other tasks in the same job .

Job : A job is a single process or a set of processes in relations which are executed concurrently for the benefit of one application . For example , a pipe-line command on a UNIX system is a job constituted of many cooperating processes .

2.3.2 Types of multiprocessing

Heterogeneous multiprocessing : In the heterogenous multiprocessing , a program is divided into many parallel sections performing completely different tasks . The tasks can be easely executed in parallel , implying a decrease of the total execution time for the algorithm . In fact , the heterogeneous tasks have so little number of messages to pass to another , that the communication paths could be slow without affection on the performance of the algorithm .

Homogeneous multiprocessing : In the homogenous multiprocessing , a job consists of several identical tasks . An application which passes most of its time in sequential code , can often be converted to a parallel version . Each paralle task do the same work , but on different datas . Some cooperation is necessary between the tasks .

2.3.3 Scheduling algorithms

The pre-scheduling : In the pre-scheduling algorithm , the processes are fixed at the compile time . In such applications , the programmer assigns a specific process to each processor . For example , the programmer can decide that the processor number 6 will be permanently affected to the management of the input/output operations , and that the processor 3 will solve the arithmetic operations .

The mixed-scheduled algorithm : The mixed-scheduling algorithm provides an intermediate step between the pre-scheduled algorithm and the passive-scheduled algorithm . In such a way to schedule , the processes are managed by a master process . All the processes execute the same code , but the master process spend a part of its time in managing the other processes . During the rest of its time , it executes the same code . The sequence for the master process is the following :

1- Distribute the code to the parallel processes
2- Execute the code as the other processes

The sequence for the other processes is the following :

1- Wait for the master process to give me some work to crunch
2- Crunch the work
3- Back to the waiting state for new work .

The passive scheduling : The passive-scheduling algorithm implies that all processes are managed by a master process . This master process spend all its time in the management of the other processes . This implies that at least 2 processes are necessary to execute any algorithm by this method . The sequence for the master process is the following :

1- Distribute the code to the processes
2- Wait that they have finished
3- Back to the step 1

For the other processes , the sequence is the following :

1- Wait for the master process to give me some work
2- execute this work
3- Return to the waiting state

The self-scheduling : The self-scheduling algorithm is more interresting because it produces what we will call 'dynamic load balancing' . The self-scheduling algorithm is valid for all the processes , and is the following :

1- Wait that some work arrives in the queue of works
2- Remove this work of the queue and execute it
3- back to step 1 , until there is nothing more to do .

The dynamic load balancing produced by this algorithm finds its interrest in the fact that all processors available are running at full time until there is no more work to do . This is the best algorithm in most cases , but we will see later that it requires certains conditions that are not always available .

2.3.4 Granularity of an application

As we discussed earlier , the granularity factor is very important for the processing of parallel algorithms . In fact , the process generation , and its termination take a great time . So , the parallel applications must be conceived so that the management times are negligible compared to the time to execute the parallel sections themselves . We will later have the opportunity to discuss about the overheads due to process generation , and termination in the PARFOR environment . Figure 2-1 shows an abstract representation of programs having a large (a) and a fine (b) granularities , respectively .

Large Granularity Program

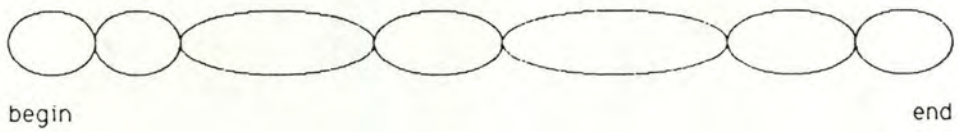


Figure 2-1a

Fine Granularity Program

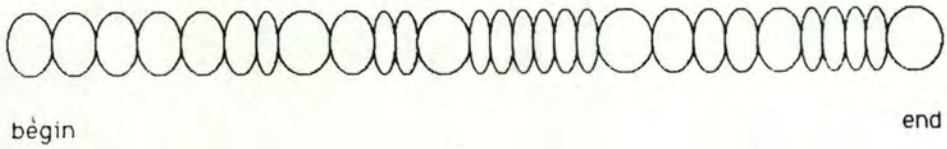


Figure 2-1b

2.3.5 Creation and termination of processes

On a SINIX system , a new process is created by a FORK() primitive . The child process is no more than a copy of the old process , using the same datas , the same registers , the same files and the same program counter . So , the child has access to all resources where the father has an access . FORK() returns the null value for the child process , and the pid of the new process for the father .

The JOIN() operation performs the opposite function to return to only one instruction flow bit it is not implemented in the UNIX environment .

The cost of a FORK() operation is high in terms of time . To reduce this cost , the parallel applications can create new FORK()s only at their beginning , and finish them only at the end of the application . Meanwhile , if a process is not necessary for a moment , it is put into a waiting queue , and its processor is released . This cost of setting into a waiting state is cheaper than the cost of a new FORK() operation . This technic is explained later and is use in the PARFOR environment .

2.3.6 Communications between processes

The UNIX system V provides some communication tools to allow communications between processes . These tools are available in the IPC system library .

The messages sysstem allows processes to communicate via messages . A message queue must be created , the structure and the type of the message defined , and a text zone reserved for the user to insert anything in it . The messages system is accessible via 2 main primitives which are MSGSND and MSGRCV . If a process is waiting for a message , its state can be defined as waiting or not .

The semaphores system allows processes to be synchronized via software locks and semaphores . The shared memory is used by this system for the definition of the queues and the structures .

The shared memory allocation system allows a user to define and use part of the memory shared among the processes . The memory is attributed and attached somewhere in the virtual space of the processes . This memory is paged like any other memory .

The most simple mechanism for communications is the shared memory . A shared space can be reserved in virtual memory at the creation date of each process . In C programs , the shared areas can be specified dynamically . In FORTRAN programs , to the contrary , the memory allocation is static and all the shared memory regions are allocated at the compile time in "common" statements .

2.3.7 Locks

A lock is a particular type of data that can have only 2 states , locked and unlocked . When a processor is intended to access a shared data structure , it must first be sure that the associated lock is unlocked . While it is not the case , it remains waiting . When the lock turns to the unlock state , indicating that no other process wants an access to the data , the processor locks the lock , access the data , then unlocks the lock . While waiting , the processor remains in a busy state .

The locks can be implemented in hardware by atomic lock memories , or simply in software , but with certain restrictions that we will explain later .

The hardware locks are called atomic lock memories because the operations necessary to acquire or release one of them are undivizable , implying that these operations can not be overlapped when many processes compete for them .

Sometimes , in certain circumstances , software locks are also interresting . These locks must be implemented in shared memory regions . They are treated just like any other memory locations .

2.3.8 Critical sections

A critical section is a section of code which must be executed only by one process at a time . Each critical section begins with a lock operation and finishes with an unlock operation . Figure 2-2 shows how a critical section can be used in a program . We can observe that only one of the 3 processes can enter the critical section at a moment .

Critical region protected by locks .

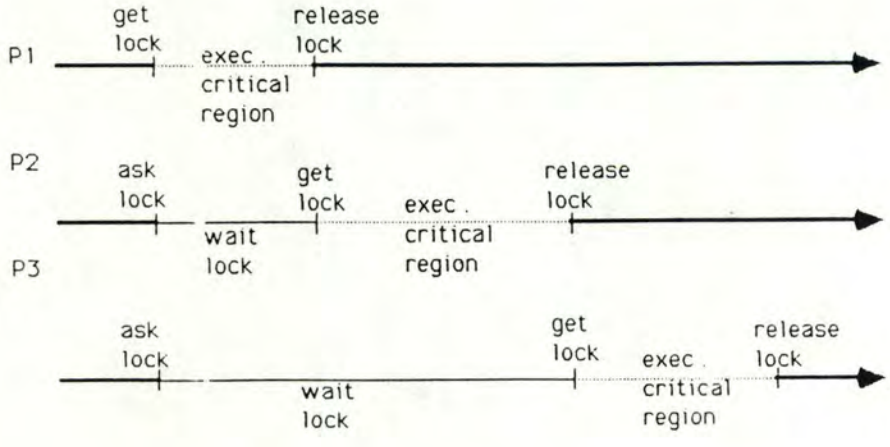


Figure 2-2

Figure 2-3 shows the standard algorithm which allows or denies accesses to a critical section of a program . If the lock associated to a critical section is locked , the process must wait that the lock turns to an unlock state .

Section Protected by Lock

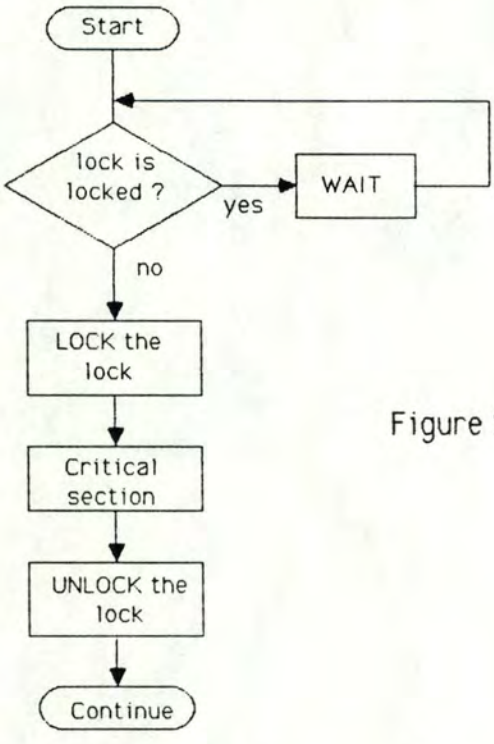


Figure 2-3

Note that a critical section can be protected by many ways , not necessarily by locks . For example , semaphores are also usefull to protect critical regions .

2.3.9 Semaphores

A semaphore is a shared data structure that can be used for the synchronizations of various coordinated processes . The most simple type of semaphore is the atomic lock . The semaphore , thus , is designed to manage the shared resources inside an application . The semaphore itself has always an integer value N . This value can be interpreted as follow :

If N > 0 , the resource is demanded N times by different processes . When the semaphore reaches the value 1 , its state becomes unlocked .

If N <= 0 , the semaphore is locked , i.e. the resource is not available , and -N represent the number of times that the waiting processes have demanded the access to the protected resource .

The value of a semaphore can be modified by only 2 algorithms . These algorithms manage the acquisition of the semaphores , and the release of them .

Acquisition algorithm :

1- Decrement N by 1
2- If N <= 0 , insert me into the processor's waiting queue and wait my tour . Else do nothing .

Release algorithm :

1- If N >= 0 , inform the first process of the processor's waiting queue that it is its tour , and shift the queue of one position . Otherwise , do nothing
2- Increment N by 1 .

2.3.10 Events

An event is something that must be waited for , before a process can start or continue its execution .An event has 2 possible values : delivered or suppressed . A process waiting for an event must wait until the event is delivered by another process . Once the event is delivered , the waiting process can continue its execution . It is the role of the master process or the role of one another process to suppress the event after its use .

2.3.11 Barriers

A barrier is a synchronization point . This point is said to be realized when it has been reached by a specified number of processes . The rule for a

process when it arrives at the barrier , is the following , in most of the standard kinds of barriers :

1- Mark me present at the barrier
2- Wait until the required number of processes at the barrier is reached
3- Reset the barrier and continue the execution .

The barriers can be implemented in many ways . The most known and used are the 2 locks barrier , and the software barrier . The implementation of both kinds of barriers have their advantages and their inconvenients .

2.3.12 Figures

The following figures show the relations between multiprogrammed system , multitasking system and multiprocessing systems .

Figure 2-4 shows how the code of a program is executed on a single monoprogrammed monoprocessor machine . The first figure (a) shows the process running on a very simple machine at a very low machine level . On the second figure (b) , one can see the intermediate use of an operating system . But the processor can still execute only one process .

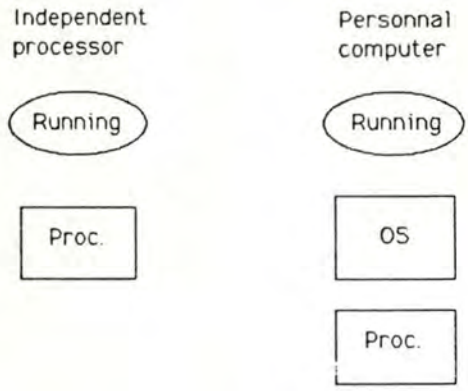


Figure 2-4a

Figure 2-4b

Figure 2-5 shows the dynamic of multiprogrammed system running on a monoprocessor machine . Many processes share the only one available process . The operating system must dispose of a scheduler to distribute fairly the processor among the asking processes . Only one process is in the running state at a time .

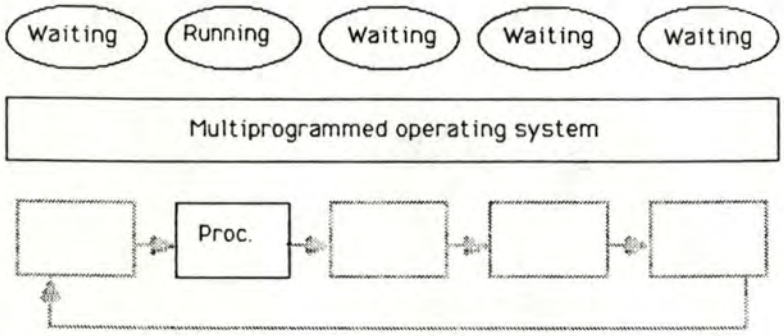


Figure 2-5

Figure 2-6 shows the dynamic of a multiprogrammed system with multiple processors . The architecture disposes of 4 processors . If more than 4 processes are introduced in the system , they must follow the rules of a multiprogrammed system , i.e. wait until the processor resource becomes available , like in the traditionnal multiprogrammed system .

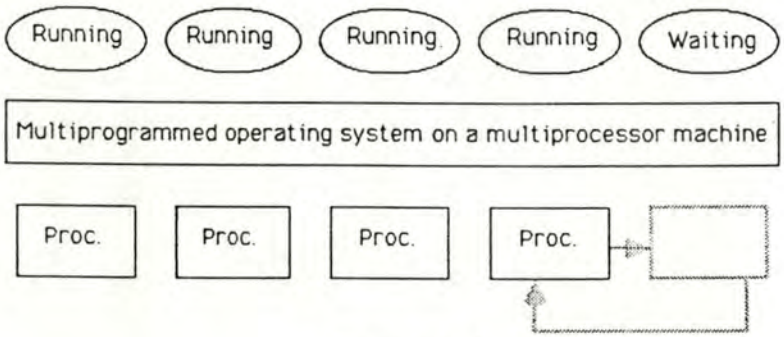


Figure 2-6

Figure 2-7 shows how a set of many processes can be executed at the same time for the benefit of one job .

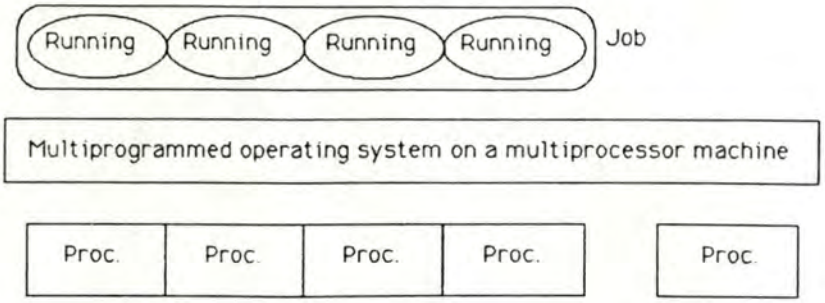


Figure 2-7

Figure 2-8 shows eventually that dynamic associations of both previous cases can appear on a computer system , for the execution of many jobs in parallel . Each of them recquires many associated processes executing the code in parallel and / or at the same time . The architecture itself disposes of many processors .

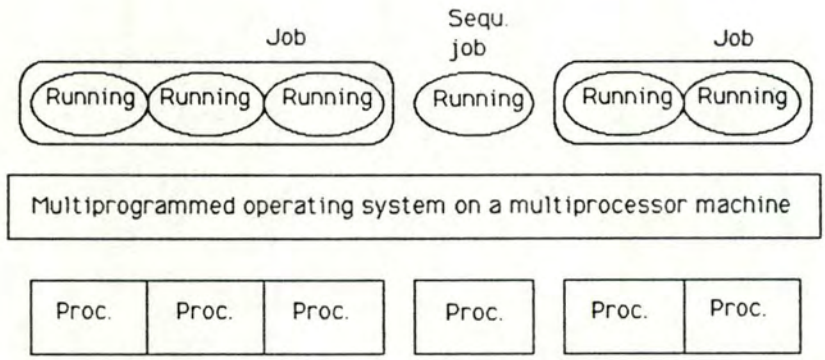


Figure 2-8

2.4 Theories concerning multiprocessor performances

2.4.1 The concept of speedup

First consider an algorithm A running on a machine which handle only one processor for the user processes . The time for the execution of the entire algorithm is T(1) . If the same algorithm A can be executed without changes on a second machine which is able to allocate N processors for the execution , then the time for the entire execution of the algorithm will be T(N) . There are relations between T(1) and T(N) .

From there , new concepts can be introduced . The is the speedup denoted by S(N) . The speedup of an algorithm is defined as the increase of speed of running an algorithm when passing from a system which executes it with one processor , to another which executes it with N processors . It is expressed as follow :

$$S(N) = \frac{T(1)}{T(N)}$$

The second concept is related to the efficiency E(N) of one of the parallel sections of the algorithm in execution . It is in relation with the concept of speedup by the following way :

$$E(N) = \frac{S(N)}{N}$$

The relation between the two concepts is that the comparison of S(N) with N is equivalent to the comparison of E(N) with 1 .

At this point , it should be pointed at , that the target of the developpement of parallel algorithms is not to increase the speedup , but well to reduce the execution time of the algorithm . In some sence , an algorithm with a greater speedup uses the hardware more intensively , but does not necessarily provide a better execution time .

2.4.2 Conflicts

We have just defined the speedup . According to [Darema-Rogers] , it is less than the linearity . In fact , the best case of speedup is a monoton increasing function of the number of processors for a given algorithm . But we will see that some controverse still remain about the speedup . This problem is a sensible point when speeking about parallelization of programs .

Conflicts seem to appear between people who are thinking about the limits of the possibilities provided by parallel algorithms . The main problem is simple to imagine . Consider for example a specific program that has a sequential execution . What could be the increase of performances given by the the same program executed with more than one processor ? The question can be reformulated in a different way : Is there an upper limit to the increase of performances due to the parallelization of the program ?

The response to this question is not trivial . Many people have different opinions about this problem which , at the first sight , seems to be simple . In fact , there is no clear response to it . So , V.FABER in [Faber-Lubeck-White] claims and demonstrates in a very simple way , that the superlinear speedup of an efficient algorithm is not possible . But in another article of the same issue of the review , D.PARKINSON in [Parkinson-86] , to the contrary , certifies that the parallel efficiency of an algorithm can be greater than unity .

This conflict among thinkers about a so simple problem is sufficient to show that some controverse still exists , and that actually , the problems highlighted by the parallel programming are not yet completely understood by the users . Of course , for the small problem evoked in the articles , there is a solution . This one is found in the default of precision in the assumptions , so that they are talking about different things . The solution

of this apparently contradiction is described in [Faber-Lubeck-White-87] and in [Janssen] of the review which introduced the original consecutive articles , to provoke confusions and reactions among readers .

2.4.3 Discussion of speedup and overheads

We have defined the theoretical concept of speedup . This concept has 2 interresting properties which are : First , this concept is processor and implementation independent , and secondly , the speedup is the notion that we are directly interrested in .

When the speedup must be measured in an algorithm , 2 critical points must be taken into account .

First , it is clear that the measurements done , must be done with the same datas for both algorithms . This implies that the same amount of work must be crunched by both versions of the algorithm , the parallelized version and the sequential version . For example , if an algorithm is designed to search a sub-string in a string , the same string and sub-string must be used in both algorithms for the measurements .

Secondly , there is a subttle point relative to the definition of speedup . Usually , the execution time of a multiprocessor algorithm executed with only one processor , T(1)' , is greater than the execution time of the execution time T(1) of the original sequential algorithm doing the same . This appears to be due to some additionnal work introduced by the parallel environment . This additionnal work is the cost of parallelization , also called overheads .

The overheads are of 2 natures .

First , the hardware overheads . These overheads are due to phenomens such as bottlenecks or communication delays . They are denoted by Hloss .

Secondly , the software overheads . These overheads are the consequence of the inaptitude of the algorithm to keep the requested processors all the time in a busy state . This is a defficiency of the algorithm . These overheads are called Sloss .

With these concepts , we can define the execution time of an algorithm on a multiprocessor system :

$$T(N) = \frac{T(1) + Sloss + Hloss}{N} + \frac{T(1)'}{N}$$

On a well designed multiprocessor system , Hloss should be negligible . But Sloss can be very great , depending on the way the algorithm has been parallelized .

According to [Darema-Rogers] , the overheads are usually low , approximately 1% of the total execution time , but it depends greatly on the kind of application studied . For example , small usual benchmarks produce overheads on systems that can take until 10% of the runtime . Normally , for the large applications , the overheads are rather small .

2.4.4 Concepts of Hockney

In a computer machine , the performance measurements can be taken by the use of some parameters . Depending on these parameters , some measures can be described . One method of measurements is proposed by Hockney in [Hockney-86] . The main principle of the method is based on the performance of a vector computer . Then the method extended to synchronization overheads due to parallel programming , and eventually to the performance measurements of a complete parallel program , depending on the number of processes executing the program . These ideas are explained in the following section .

2.4.5 The main idea : the performance of a vector

Considering a vector operation , it can be executed in a time T , which can be decomposed into 2 sub-times ; a fixed time To consisting of the time necessary for initialization , a variable time depending on the number N of elements in the considered vector , and on the time Te necessary to compute one element of the vector . This is described by the formula :

$$T = To + (N * Te)$$

From this expression , we can compute the mean time Tem of computation of one element in the vector :

$$Tem = \frac{T}{N} = \frac{To + (N * Te)}{N} = \frac{To}{N} + Te$$

From there , we can describe the mean rate of computation r of an element , also called performance . It is expressed as

$$r = \frac{1}{Tem} = \frac{1}{(To / N) + Te}$$

Now consider ri as the maximum performance that can be achieved by the vector , i.e. the theoretical performance of an infinite vector length for the machine . This can be described by

$$ri = \lim_{N} \frac{1}{(To / N) + Te} = \frac{1}{Te}$$

Note however that this performance is never reached because it is valuable for a vector of infinite length . It can only be approached by large vector lengths .

Let us also define N1/2 as the length of the half performance , i.e. the length of the vector , which is necessary to reach the half maximum theoretical performance of this vector . So we have the following definition :

$$N1/2 = ri * To = To / Te$$

One can check that if To is null , then N1/2 = 0 ; and that if To is infinite , then N1/2 is also infinite .

The time T for performing the entire vector operation can be rewritten

according to these concepts as

$$T = To + (N * Te)$$
$$= Te * [(To / Te) + N]$$
$$= (1 / ri) * (N1/2 + N)$$

and the performance of the vector can be redefined as

$$r = \frac{1}{Te + (To / N)} = \frac{1}{[1 + (To / (Te * N))] * Te}$$

$$= \frac{ri}{1 + (N1/2 / N)}$$

We can also simplify the expression of r by defining a function pipe(x):

$$pipe(x) = \frac{1}{1 + 1/x}$$

So we rewrite r as

$$r = ri * pipe(N / N1/2)$$

Wy these notions of ri and N1/2 ? It seems that they are more convenient for the user to compare directly the performance r of his vector with the theoretical maximum performance of it . In the same sence , N1/2 allows the user to compare the length of his vector with the length of the half performance .

Figure 2-9 shows a synthesis of these relations and a graphical representation for them .

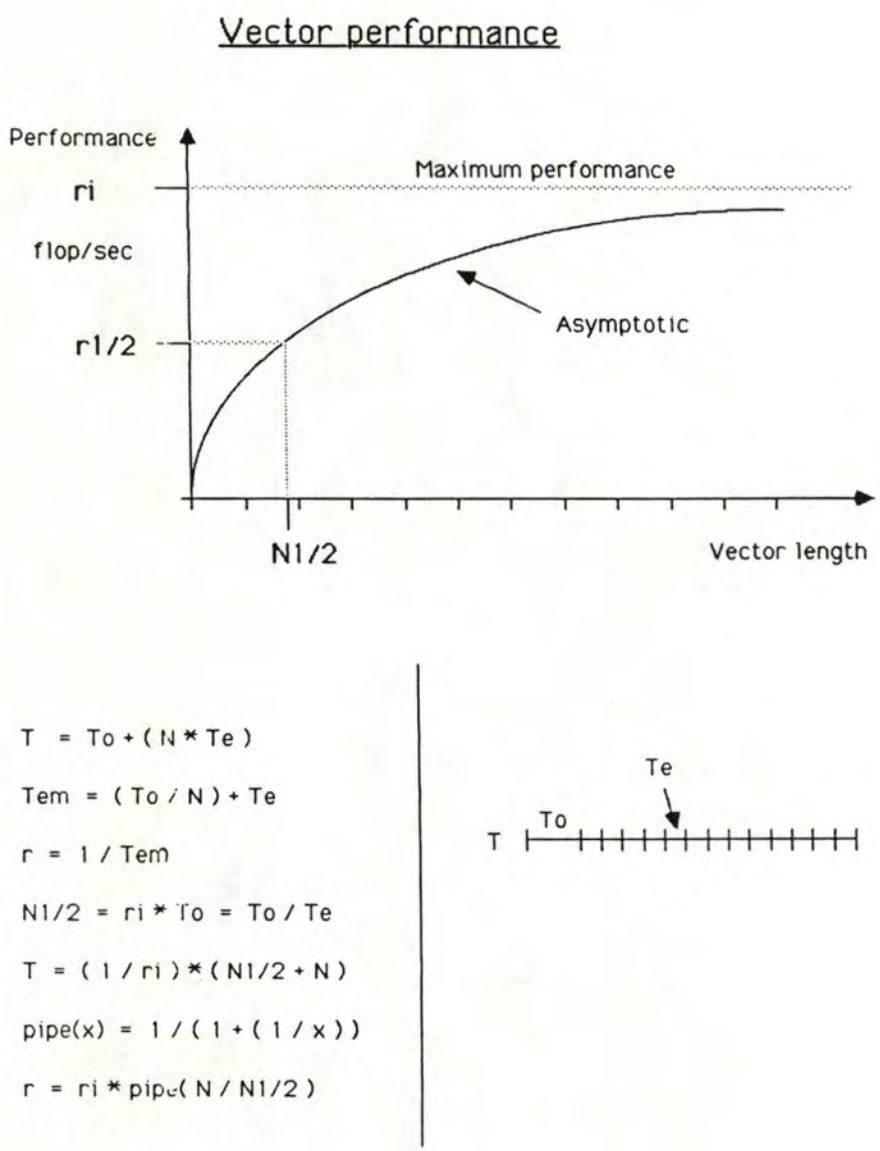


Figure 2-9

2.4.6 Synchronization overheads

We have widely explained the theoretical concepts of performance for vectors although it is not directly related to our problem . However , using these basic concepts , Hockney extends them for measuring the synchronization overheads which occur in parallel branches of a parallel program .

If we have a simple parallel environment executing various branches of a parallel program using primitives such as FORK and JOIN , we can consider each branch of such a parallel program as a sequence of serial sections . The execution time of one parallel branch of the program can be described as follow with the concepts used in the vector measurements :

$$T = (1 / ri) * (S + S1/2)$$

In this expression , we use the concept of ri introduced earlier for the vector performance measurement . It has here the same signification as we gave for the vector measurement , i.e. , the maximum performance of an operation within a sub-section of a parallel section .

Two other new concepts are introduced . The first of these is S , representing the computation work measured inside one parallel section , in terms of equivalent floating point operations . This concept has its corresponding N in the vector measurements that we described in the previous section .

The second new concept is S1/2 . S1/2 is the synchronization overhead in terms of quantity of work , expressed in equivalent floating point operations , which could be executed during the synchronization time . In other words , S1/2 is the quantity of work , in a parallel section , which is necessary to reach the half asymptotic performance . So , it would be very usefull to create a parallel section in a program if the equivalent work implied to create it , is clearly less than S1/2 . S1/2 is defined as N1/2 was defined for the vector performances .

In a previous chapter , we also described the notion of granularity of a program . Now , we can say that S represents this granularity factor , and S1/2 represents the minimal granularity for a parallel section .

As we can see , there is some parallelism between the performances of a vector and the performances of a parallel section of a program . We can complete our equations like we made for the vector performances :

We had

$$T = (1 / ri) * (N + N1/2)$$

and

$$r = \frac{ri}{1 + (N1/2 / N)} = ri * pipe(N / N1/2)$$

Then we have also :

$$T = (1 / ri) * (S + S1/2)$$

and

$$r = \frac{ri}{1 + (S1/2 / S)} = ri * pipe(S / S1/2)$$

for the synchronization overhead problem , in one branch of a parallel program .

2.4.7 Parallel program performance

In a similar way , we can describe the concepts related to the performance of a program .

For this , we consider that the time Tp to execute entirely a program is the sum of the time To spent in the sequential sections , and the time T1 spent in the parallel sections , devided by the number P of parallel sections . So we have :

$$Tp = To + (T1 / P)$$

We can also define the execution rate Rp as

$$Rp = \frac{1}{Tp} = \frac{1}{To + (T1 / P)}$$

$$= \frac{Ri}{1 + (P1/2 / P)} = Ri * pipe(P / P1/2)$$

where Ri is the maximum performance of the program , increasing asymptotically with the number of parallel sections . In fact , the factor T1/P diminishes , leading the performance of the program to the performance of the unreductable section having a time To . Ri is defined as

$$Ri = 1 / To$$

P1/2 is the number of processes necessary to reach the half performance of the program (considering that a processor is given to each parallel section) . It is defined as :

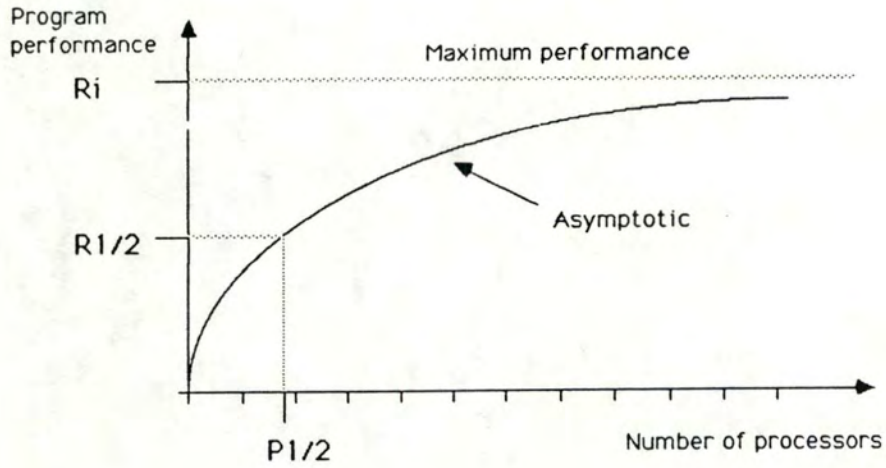
$$P1/2 = T1 * Ri = T1 / To$$

Compared with the vector performance , the program performance uses little differences in formulas because the critical concept in a parallel program is not the same as in a vector . In vectors , this critical concept is To , the initialization time of the vector operations , while here , in a parallel program , the critical concept is T1 , the execution time of a parallel section .

Figure 2-10 shows a synthesis of these relations and a graphical representation of them .

Program performance

For this graphic , it is assumed that the synchronization times are null or negligible .



$$Tp = To + (T1 / p)$$

$$Rp = 1 / Tp$$

$$= Ri / (1 + (p1/2 / p))$$

$$Ri = 1 / To$$

$$P1/2 = T1 * Ri = T1 / To$$

$$Rp = Ri * pipe(p / p1/2)$$

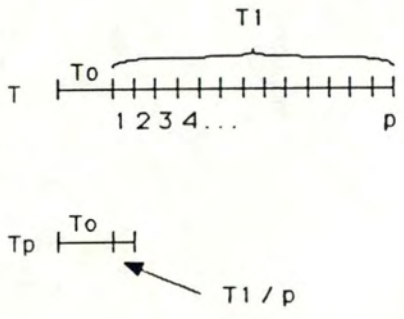


Figure 2-10

2.4.8 Modelization of a parallel program with synchronization overheads

We have described the performance of a theoretical parallel program considering that Rp increases always , but more and more slowly in an asymptotic way . However , if the number of processors becomes larger , the synchronization overheads also increase , leading Rp to a maximum R' , to the contrary with Rp .

The behaviour of such a situation can be described by the following function , which has been derived from a lot of observations by Hockney :

$$Rp = \frac{Ri}{1 + \left(\frac{P1/2}{P} * \left\{ 1 + \left[\frac{1}{n-1} * \left(\frac{P}{P'} \right)^{n} \right] \right\} \right)}$$

This model of performance shows that , when the number P of parallel sections of a program increases - or the number of processes - , the synchronization overheads increase also quickly . This implies that there is an upper limit to the performance of a parallel program , and a drop of performances when the number of processes increases further than a certain number at which the maximum of performance is reached .

In the model , we can see that the velocity of increase of the synchronization overheads with P is determined by n , which is called the the index of synchronization . This index should be as low as possible . The magnitude of the synchronization overheads is determined by P' . So, more P is large , lower the overhead is .

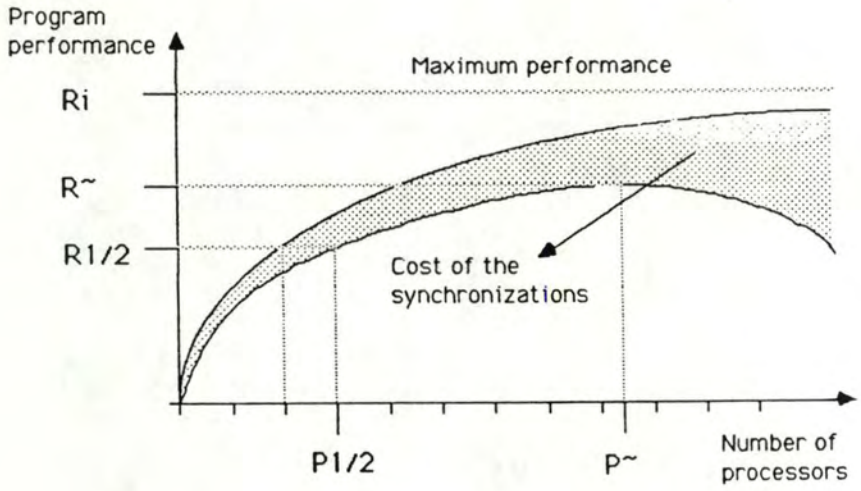
The maximum performance Rp of the program , in the model , is reached when P = P' , and then , the expression Rp turns to R' which is

$$R' = \frac{Ri}{\left[1 + \frac{n}{(n-1)} * \left(\frac{P1/2}{P'} \right) \right]}$$

Figure 2-11 shows a synthesis of these relations and a graphical interpretation for them .

Program performance

For this graphic , it is assumed that the synchronization times are not necessary null .



$$Rp = \frac{Ri}{1 + \left(\frac{p1/2}{p}\right) * \left\{1 + \left[\frac{1}{n-1} * \frac{pEn}{p\tilde{}}\right]\right\}}$$

Factor of synchronization

The peak appears at $p = p\tilde{}$, with

$$R\tilde{} = \frac{Ri}{1 + \left\{\frac{n}{n-1} * \frac{p1/2}{p\tilde{}}\right\}}$$

Figure 2-11

2.5 Practical measures in a parallel program

2.5.1 Benchmarks in a multiprocessor environment

Our final aim is to make measures on the MX500 within the PARFOR environment , and later , with FORCE .

A benchmark program is constructed to test the machine in a very tight domain . But the results produced by the benchmark are not only the fact of the machine , they are greatly dependent of the environment . This implies the capacities of the compiler which has compiled the benchmark , the code of the benchmark , the operating system in which it is executed , the load of the system , the precision of the tools used for measuring inside the benchmark and inside the operating system , and the measurement technics used .

While trying to parallelize a benchmark , more than for a program , a choice of the level at which it will be parallelized must be done . In fact , a benchmark is often a synthetic program , repeating very often a certain amount of tasks differing by the datas they use , or do not differ any way . This seems to be ok for the sequential environments . But when trying to parallelize these benchmarks for their execution on a parallel machine , the question of the level at which it must be parallelized is more important .

For example , taking a benchmark program solving linear equations systems , it can be parallelized at various levels . The highest level consists in parallelising the number of times that the same system is solved in the outermost loop of the benchmark . A second possibility is the parallelization in the main parts of the algorithm . And a third level would be the parallelization of the innermost routines which are the effective crunchers of the algorithm .

The first possibility will take into account that the benchmark is a synthetic and repetitive job having no interrest , except that it provides

time results concerning the calculation rates of the computer on which it is executed . This is the outermost level . At this level , because of the repetitivity of the benchmark , it is relatively easy to parallelize so that the processes are well distributed on the set of processors .

The second possibility takes into account the particularity of the algorithm . So , at this level of parallelization , the algorithm itself is parallelized , but not the benchmark . This means that the benchmark is still executed in a sequential way , but the main parts of the algorithm called many times is distributed on the processors .

The third possibility is the lowest level of parallelization . In this method , the lowest level routines of the algorithm are parallelized . These routines are simple , but the parts of the algorithm spend most of their time in their execution . So their parallelization seems to be interresting . But the problem is that these routines are really small , and the overheads produced by the use of the parallel ennvironment are rather great compared to the small work crunched by these routines .

These possibilities of parallelization could be done simultaneously , but our parallel environments do not accept this . Only one level of parallelization is accepted in a program . The choice of the level depends essentially on the size of a parallel section , the granularity of the program . As we said earlier , the various overheads due to the effects of parallelization must remain very low or negligible compared to the work performed by the section .

2.5.2 Time measurements

The most difficulties in trying to measure the performances of a machine of the performances of a program , remain certainly the problem of taking the measures themselves . Taking the right measures at the right points in the programs involves a certain number of questions and requires many choices .

The reasons of this are of various natures .

The first reason is that there exists no global theory or method for trying to measure the performances of parallel applications (which can be the system) . There exist partial theories and methods that are valid in only particular situations . However, when we are determined to take measures , the environment itself is fixed , and the question of the particularity seems not to be critical . But in most cases , the measures to be taken , are destinated to be compared with other measures (for example , if we measure the abilities of a FORTRAN compiler on a particular machine , it would be usefull to compare directly the results of these measures with those provided by the same compiler working on another machine . But for this , the tools for measuring should be the same on both machines . Is it the case in the reality ?) . So , for this reason , the measures should be comparable , i.e. they should have been taken according to similar method on both systems . This is a large problem in taking measures on computers .

A second reason of these difficulties , is that of the choice of the method to follow . This could be a subquestion of the first reason . In the area of measures , many articles and books have been written . And sometimes , some of them lead to completely different approaches for a given problem , leading to results that can not be compared , as explained earlier . Sometimes , some difficulties appear . It is possible that a method is followed at the very beginning of the measures , and that this method must be changed in favour of another later , because of some critical reasons that only appear late in the developpement of the tests .

Another reason of the problems is that of the availaibility of the tools necessary to implement a particular method on the system to measure . After the choice has been taken , the question is the "how to do it " . If the method has been choosen according to the tools available , this is not a problem , but maybe the method has not well been choosen . On the other hand , if the method has been choosen according to the necessities of the problem to be solved , then , the availaibility of the tools can be a great problem . It can involve the building of new tools that were not originally availaible on the system to test .

A fourth reason of the difficulties involved by the real measurements is the way the tools are implemented on the system to test . Normally , a system is provided with some basic tools . The problem is to know exactly

what the routines perform , when they are called to take measures . In fact , the time routines provided and the standard libraries seem not to be sufficient to allow the creation of a set of tools . The way and the environment in which they are conceived are also necessary , to know and understand exactly what they measure . Otherwise , the measures taken are greately undefined , and the results remain relatively uncertain . It seems to be necessary to know the complete implementation of them , until the hardware level , the original source of informations concerning the time in a computer .

A fifth reason of the difficulties involved , is that a real environment involves errors . The problem becomes of great importance when trying to compute statistics concerning the measures taken . Those measures , often because of certain hidden reasons , may contain values that apparently do not match with most of the other values , taken in the same way in the same circumstances . The question is "what to do with these seemingly error values ? " . Should they be taken into account or should they be discarded from the set of results ? The problem is that these values have been taken in the same conditions than the others . So perhaps they are good , perhaps not , but it is uncertain about them . Sometime too , the discovering of errors in measures can lead to the rediscussion of the validity of the tools from which the values are taken . From there , the necessity to know the exact implementation of the time routines as discussed above . So , we think that the problem of errors must take place inside the discussion concerning the measures of performances of a system .

A sixth reason is the variability of the measurement results taken . This can be a generalization of the problem of errors , except that the values we talk about seem to be correct values . One problem is that the results are different for multiple executions of the same programs in the same environment (including the load of the system) . Another problem is that the results also vary , depending on the load and the time at which the measures are taken (some processes , for example, are activated only when a "sufficient" amount of work is present in the queue . So , depending on the time at which the work arrives , the load on the system also varies) . This kind of variability is sometimes called "statistic errors" . This involves that the reliability of the measures should be envizaged in further measurements , and that studies concerning the trust of the

measures should also be reported . It is sometimes the case that reports claim in favour of the high performances of an application , showing numbers , but discarding informations concerning their reliability and the context in which they have been taken .

2.5.3 Cpu-time and Real-time

In sequential applications , the measures taken are usually the cpu-time and the real-time . The cpu-time is more system oriented , while the real-time is more user oriented .

The cpu-time measures the cpu-time consumption of the application . The real-time measures the time that the user had to wait for its application to be finished , and get the results at his screen .

In parallel systems , it is not so true to make the distinction between the application and the parallel environment . Both are tighly coupled . Then , what about the distinction between the user point of vue , and the system point of vue ? They are quiet associated .

However , to our point of vue , we continue to think that the concepts are valid in parallel environments , even if the notions are not so clear . We keep this attitude because we have no other way to take measures , and because the abstract concepts remain the same .

2.5.4 Time routines

For the time measurements we make in the PARFOR environment , we use the time functions provided by UNIX . So , all measures taken have the precision of the functions provided by UNIX . These routines are only 2 and are the following :

TIMES()
CLOCK()

Both routines are part of the standard UNIX environment in the system V .

The first routine , TIME() , fills its arguments with time-accounting informations . These informations come from the calling process . The informations provided are the user-time and the system-time .

The user-time is defined as the cpu-time used while the processor is executing instructions in the user space of the calling process .

The system-time is defined as the cpu-time used by the system on behalf of the calling process .

This routine TIMES() also provides as its return value , the elapsed time from an arbitrary point in the time . Because of this , it is necessary for the exploitation of this information , to call at least 2 times the routine TIMES() to compute the difference .

All informations provided by TIMES() are given in CLK_TICKths of a second .

The second routine , CLOCK() , returns the amount of cpu-time used since the first call to clock . The time reported is the sum of the user and system times of the calling process .

The value is returned in microseconds . Note that normally , this value should be the sum of the user and the system times . In the reality , because of the better precision of the CLOCK() function , there are variations , but little . That is the reason why we take this measure into account .

Both functions are called in a C procedure called ZEIT() providing these values in miliseconds , and in microseconds for the cpu-time .

The ZEIT() function has 4 parameters , which are the following :

1) Real-time in miliseconds
2) User-time in miliseconds
3) System-time in miliseconds
4) Cpu-time in microseconds .

The interface is standard , so that ZEIT() function can be called very easely from a C program or a FORTRAN program .

2.5.5 One representation of a parallel program

A standard simple parallel program is made of 3 main parts . A standard sequential part for declarations and initializations , a set of parallel sections depending of the application in which the main treatement of the algorithm is made , and the termination part in which all results are collected and the final results produced .

This is very simple . But often this frame is repeated many times , a serial section , a parallel section , a serial section again , a parallel section again , and so on . . . But the main principle remains the same .

Figure 2-12 shows this representation .

Frame of a PARFOR Program

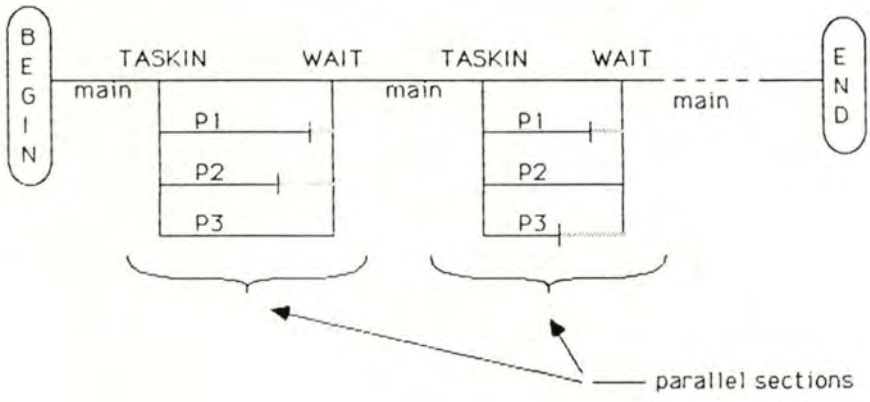


Figure 2-12

2.5.6 Test-points in a parallel program

Our time measurements in the parallel program are made at certain places . Each test-point has a number . With 7 test-points in a parallel program or part of a parallel program , we can measure all interresting times . The test-points are the following :

Test-point 1 is made at the begin of the program itself , just after the declaration part of the FORTRAN program .

Test-point 2 is placed at the end of the first serial section of the application . The point is just before that the parallel sections are initiated .

Test-point 10 is placed at the begin of the parallel section , i.e. it is the first statement of the child task . Note that this test-point 10 is repetitive because appearing in each parallel section .

Test-point 11 is placed at the end of the paralle section , i.e. it is the last statement of the child process . This test-point 11 , as the complementary to the test-point 10 , is also repetitive for the same reason .

Test-point 3 is located just after the synchronization barrier statement in the main task . At this place , all children have finished their execution , and the synchronization barrier is passed .

Test-point 4 is placed at the end of the sequential termination section of the application . This is the logical end of the application .

Test-point 5 is placed just after the test-point 4 . It is placed there to measure the time consumed by the time routine itself .

Figure 2-13 below shows these time measurements in a PARFOR program .

Time measurements in a PARFOR program

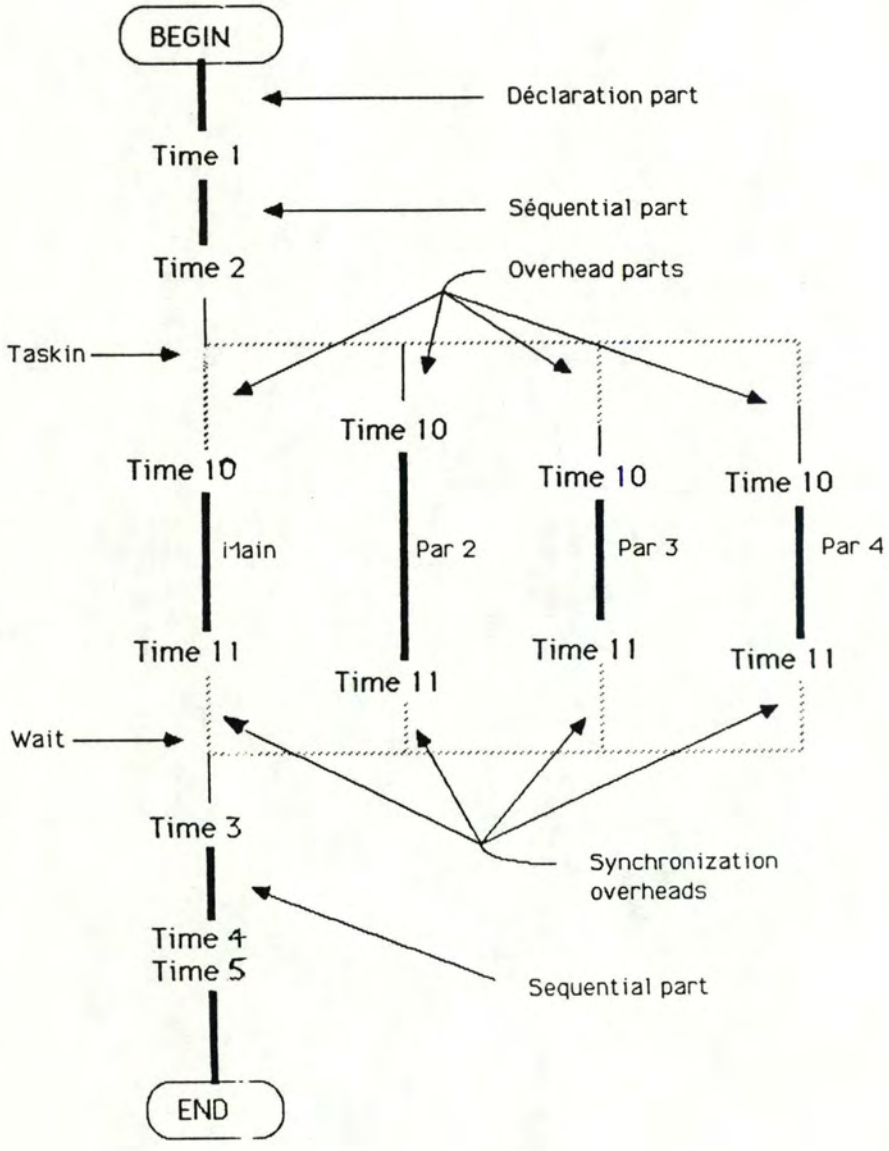


Figure 2-13

2.5.7 Description of a tool that uses these test-points

The time measures are computed after the application is terminated . To facilitate these computations , we have created a FORTRAN subroutine that takes as parameters , the times measured at the test-points described above , and performs all interresting computations on these times .

As we have said before , we have 7 test-points . For each of them , we have 4 values recorded : The real-time , the user-time , the system-time , and the cpu-time . This is valuable for the test-points 1 , 2 , 3 , 4 and 5 . For the test-points 10 and 11 , we need to record the 4 values for each parallel section . So , we need for them , a table containing at least a number of entries greater than the number of processes . But , because of the static allocation of memory of the FORTRAN systems , we fix an arbitrary large value for them , to be sure that it will be sufficient in most cases .

We describe now the measures computed on the basis of the times recorded at the test-points , and the names of the variables we used for the results . We present them for the general case . So , in the names of the variables , when we use the letter X , it must be replaced by R for the real-time , U for the user-time , S for the system-time , and C for the cpu-time .

These are the following :

- The time of the time routine which is used to take the times . This measure is computed on the basis of test-points 4 and 5 . We have

  ```
  xt = xt5 - xt4
  ```

- The time of the initialization section of the application . This time is computed on the basis of the test-points 1 and 2 . We have

  ```
  xtinit = xt2 - xt1 - xt
  ```

- The time of the termination section of the application . This time is computed on the basis of the test-points 3 and 4 . We have

```
xtterm = xt4 - xt3 - xt
```

- The total time spent in the sequential sections of the application . This time is computed on the basis of xtinit and xtterm . We have

```
xtsequ = xtinit + xtterm
```

- The time passed during the global parallel section of the application . This time is computed on the basis of the test-points 3 and 2 . We have

```
xtpara = xt3 - xt2 - xt
```

- The global time of the application . This is based on the sequential and parallel times . We have

```
xttota = xtsequ + xtpara
```

- The total time of the application computed in another way . This measure is based on the base of test-points 1 and 4 . We have

```
xtglob = ( xt4 - xt1 ) - ( 3 * xt )
```

- The execution time of each parallel section . This measure is based on the test-points 10 and 11 , and is repetitive for each parallel section . We have

```
xtpar(i) = ( xt11(i) - xt10(i) ) - xt
```

- The initialization overhead of each parallel section of the application . This measure is based on the test-points 10 and 11 and is repetitive for each parallel section . We have

```
oxtpi(i) =  xt10(i) - xt2 - xt
```

- The termination overhead of each parallel section due to the end of the section and to the wait for synchronization of all processes to finish . This measure is computed on the basis of test-points 11 and 3 and is repetitive for each parallel section of the program . We have

oxtpt(i) = xt3 - xt11(i) - xt

- The total overhead for each parallel section . This is computed on the basis of the initialization and termination overheads , and is repetitive for each parallel section . We have

  soxtp(i) = oxtpi(i) + oxtpt(i)

- The mean time passed in the parallel sections of the application , included the main task . This is computed on the basis of all execution times in the parallel sections , without overheads . We have

  mxtpar = (xtpar(1) + xtpar(2) + . . . + xtpar(n)) / n

- The mean time for the initialization overhead of the parallel sections . This is computed on the basis of each initialization overhead , including the main process . We have

  moxtpi = (oxtpi(1) + oxtpi(2) + . . . + oxtpi(n)) / n

- The mean time for the termination overhead of the parallel sections . This is computed on the basis of each termination overhead , including the main process . So , we have

  moxtpt = (oxtpt(1) + oxtpt(2) + . . . + oxtpt(n)) / n

- The mean time of the total overhead in the parallel sections . This is computed on the basis of each total overhead in the parallel sections of the program , including the main process . So , we have

  moxtp = (soxtp(1) + soxtp(2) + . . . + soxtp(n)) / n

2.5.8 Second kind of tests : speedup and efficiency

The concepts of speedup and efficiency have been introduced in a previous section . In this section , we introduce the basic measures that we make for the performance measurements .

In this way to take the measures and compute the results , the final calculations are based on many executions of the algorithms . This implies that the results provided are relative from one execution to another instead of beeing absolute . This new type of measures is more general , and more user oriented , as we will see later .

We can also confess that these tables were designed after the observation that the results provided by the previous measurements were sometimes false . This was mainly due to the bad precision of the time routines , compared to the relatively small times that are recorded for the overheads to measure .

2.5.9 Description of the tools and the result tables

We present some results of the tests in tables . A table is a page . Each page is related to the execution of a test program with a number of processes varying from 1 to 10 , and other fixed parameters .

The fixed parameters are dependent on the tests made and are therefor described for each kind of test .

Each table contains 10 sub-tables . Each sub-table provides results for the execution of the program with a fixed number of processes , between 1 and 10 . We choose this interval because the machine on which we make the tests is configured with 6 processors .

Each sub-table contains 11 columns , having the following meanings :

Column 1 : Type of the algorithm

The measures are taken in tables . For calculating the various results , we always base our measures on 3 executions of algorithms in different conditions .

The principle is that all the times are recorded at the run time of the various algorithms . The calculations are executed after the algorithms are finished .

1- Sequential :
The first execution is the sequential algorithm . This algorithm is the original unmodified sequential version of the program that has to be parallelized . This first algorithm is always the referential case . All comparisons of the parallel versions are done on the basis of the time consumptions of this version .

2- Parallel with 1 process :
The second execution is the parallelized version of the original sequential version , but executed with only 1 process . This version is executed to have a global idea about the amount of overheads produced by the activation of the parallel environment , and by the way that the sequential algorithm has been parallelized . This execution is important for the comparisons with the sequential case , and for determining the amount of overheads due to the use of this parallelized version executed sequentially , compared with the purely sequential original version .

3- Parallel :
This algorithm is exactly the same as the second algorithm , but is executed with the specified number of processes indicated at the top of the column . The version is the normal parallelized version running on the multiprocessor system .

Column 2 : Flops

This column provides the estimated number of floating point operations that are performed to complete the algorithm . This number is computed according to the formula

Flops = < depends on the algorithm to measure >

Column 3 : Flop-rate

This column provides the estimated flop-rate of the couple machine-algorithm . This flop-rate is calculated on the basis of the number of floating point operations done (column 2) and the cpu-time consumed to perform them (column 4) . The formula is the following :

$$\text{Flop-rate} = \frac{\text{Flops}}{1.0\ E6\ *\ \text{cpu-time}}$$

The factor 1.0 E6 is inserted to provide the results in Mflops instead of in flops .

Column 4 : Cpu-time

This column provides the cpu-time consumed and measured in the entire algorithm . It includes the user-time and system-time . It is provided in seconds .

Column 5 : Speedup

This column provides the speedup in terms of the cpu-time . This speedup is measured on the basis of the sequential algorithm , and is computed as follow :

$$\text{Speedup} = \frac{\text{Cpu-time sequential}}{\text{Cpu-time parallel 1 process}}$$

or

$$\text{Speedup} = \frac{\text{Cpu-time sequential}}{\text{Cpu-time parallel}}$$

Column 6 : Efficiency

This column provides the efficiency in terms of cpu-time . It is computed on the basis of the speedup (column 5) , and the number of processes used to execute the algorithm as follow :

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of processes}}$$

Column 7 : Real-time

This column provides the real elapsed time consumed to perform the entire algorithm .

Column 8 : Speedup

This column provides the speedup in terms of real-time . It is computed as follow :

$$\text{Speedup} = \frac{\text{Real-time sequential}}{\text{Real-time parallel 1 process}}$$

or

$$\text{Speedup} = \frac{\text{Real-time sequential}}{\text{Real-time parallel}}$$

Column 9 : Efficiency

This column provides the efficiency in terms of real-time . It is computed in the following way :

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of processes}}$$

where the speedup comes from column 8 .

Column 10 : User-time

This column provides the user-time . This time is part of the cpu-time . It is the time passed in the user defined procedures . It is provided in seconds .

Column 11 : System-time

This column provides the system-time . This time is part of the cpu-time . It is the time spent in the system calls .

2.5.10 Scheme of the measures

Time(1)

.

.

Execution entire algorithm

.

.

Time(2)

.

.

All performance computations

Figure 2-14 below shows a standard table containing execution results of a test program for this method of taking measurements .

Tests with the SAXPY routine , single precision

Leading dimension : 10000
Vector dimension : 4000
Parallel threshold : 100
Execution number : 1000

1 process	Flops	Flop-rate	Cpu-time	Speedup	Efficien.	Real-time	Speedup	Efficien.	User-time	Sys-time
Sequential :	8000	0.0491	0.1628	1.0000	1.0000	0.1620	1.0000	1.0000	0.1501	0.0127
Par. 1 proc. :	8000	0.0490	0.1634	0.9967	0.9967	0.1630	0.9939	0.9939	0.1556	0.0078
Parallel :	8000	0.0490	0.1632	0.9978	0.9978	0.1630	0.9939	0.9939	0.1556	0.0076

2 processes	Flops	Flop-rate	Cpu-time	Speedup	Efficien.	Real-time	Speedup	Efficien.	User-time	Sys-time
Sequential :	8000	0.0490	0.1632	1.0000	1.0000	0.1630	1.0000	1.0000	0.1497	0.0135
Par. 1 proc. :	8000	0.0490	0.1634	0.9989	0.9989	0.1630	1.0000	1.0000	0.1561	0.0073
Parallel :	8000	0.0924	0.0865	1.8852	0.9426	0.0860	1.8953	0.9477	0.0820	0.0045

3 processes	Flops	Flop-rate	Cpu-time	Speedup	Efficien.	Real-time	Speedup	Efficien.	User-time	Sys-time
Sequential :	8000	0.0492	0.1626	1.0000	1.0000	0.1630	1.0000	1.0000	0.1501	0.0126
Par. 1 proc. :	8000	0.0489	0.1635	0.9951	0.9951	0.1640	0.9939	0.9939	0.1562	0.0072
Parallel :	8000	0.1168	0.0685	2.3750	0.7917	0.0690	2.3623	0.7874	0.0597	0.0088

4 processes	Flops	Flop-rate	Cpu-time	Speedup	Efficien.	Real-time	Speedup	Efficien.	User-time	Sys-time
Sequential :	8000	0.0490	0.1633	1.0000	1.0000	0.1630	1.0000	1.0000	0.1502	0.0131
Par. 1 proc. :	8000	0.0497	0.1610	1.0143	1.0143	0.1610	1.0124	1.0124	0.1557	0.0052
Parallel :	8000	0.1441	0.0555	2.9400	0.7350	0.0560	2.9107	0.7277	0.0479	0.0076

5 processes	Flops	Flop-rate	Cpu-time	Speedup	Efficien.	Real-time	Speedup	Efficien.	User-time	Sys-time
Sequential :	8000	0.0486	0.1644	1.0000	1.0000	0.1650	1.0000	1.0000	0.1503	0.0142
Par. 1 proc. :	8000	0.0496	0.1614	1.0190	1.0190	0.1620	1.0185	1.0185	0.1557	0.0057
Parallel :	8000	0.1974	0.0405	4.0572	0.8114	0.0410	4.0244	0.8049	0.0365	0.0041

6 processes	Flops	Flop-rate	Cpu-time	Speedup	Efficien.	Real-time	Speedup	Efficien.	User-time	Sys-time
Sequential :	8000	0.0473	0.1691	1.0000	1.0000	0.1700	1.0000	1.0000	0.1464	0.0227
Par. 1 proc. :	8000	0.0486	0.1646	1.0276	1.0276	0.1650	1.0303	1.0303	0.1559	0.0086
Parallel :	8000	0.2116	0.0378	4.4741	0.7457	0.0380	4.4737	0.7456	0.0346	0.0032

Figure 2-14

2.6 Parallelization of applications

2.6.1 Technics to parallelize applications

There exist 2 main types of technics to parallelize applications :

1) Technics going backwards

- Detection in applications of parallel tasks
- Modification of the processing order of some parts of the application

2) Technics more radical going forwards

- Asynchronous processing
- Domain decomposition
- Operator decomposition
- Pipeline processing

Detection of parallel tasks

This is not really a technic by itself . Rather , it should be considered as a forced first step for every type of parallelization technic that is to be applied later . This step is relatively easy for some applications that are loosely coupled , but can be very difficult for applications that are thightly coupled (where many synchronizations are necessary) .

Modification of the order of processing

This kind of parallelization of an application consists in taking various parts of an the application and arrange them into another order in such a way that their execution can be done in parallel , without affecting the results of the original program . Only the time of the execution , due to these moves of treatements or parts treatements , is decreased . This is

not a true thing to find the portions of application which can be moved , and less trivial to move them to their correct place . This type of parallelization is greatly dependent on the type of application that is treated .

Asynchronous treatement

This method consists in taking the application and in trying to find in it many treatements that can be distributed in several processes working asynchronously , with a minimum of interactions between them . So , the 2 main rules of this technic are 1) minimize the synchronizations between the processes , and 2) , increase at the maximum the number of processes that can be executed in parallel .

Domain decomposition

This technic consists in detecting the integration domain of the application and divide it into a set of sub-domains , allowing by this way the assignation of the original task to many processes corresponding to the number of sub-domains detected . Most of large scientific applications are designed to modelize physical processes in the space and time doamains . This implies that applications are often regular and repetitive . In this area , the method takes all its advantages because of the relatively easyness to decompose the original problem into a number of sub-problems , working with different datas . A simple example is the large matrix operations .

Operator decomposition

This technic is based on the use of multiple processes to compute simultaneously different operators on different datas . But this approach is very bounded to the particular hardware configuration . In particular , it seems to be essential that the load on the processes is well balanced to take the maximum advantage of the technic . Indeed , the target is to maximize the use of the resources and to minimize the bad effects of synchronizations between the processes .

The pipeline processing

Thie pipeline processing consists in executing an application with many processes . It is based on the principle of the coroutines executed in parallel . Each coroutine takes data structures from an entry , modifies it and puts the result data structure to its output . So , a chain of processes can be executed in parallel using this technic , each process taking for its entry , the result or output of the previous process in the chain . This type of parallelization is better designed for commercial applications , but can also be used within scientific domains .

2.6.2 The effort in trying to parallelize an application

The human effort is an important factor to take into account when trying to parallelize an application . This effort is greatly dependent on several parameters :

- The complexity of the code of the application
- The familiarity of the paople trying to parallelize , with the nature of the physical problems
- The familiary of the person with the algorithms used in the application
- The modifications made in the algorithm of the application
- The facilities provided by the environment in which the parallel program is to be designed .

2.6.3 Problems when programming in a parallel environment

When working on a sequential computer , it is relatively easy to create , debug and maintain applications . In such a sequential environment , everything is deterministic , so that it is always possible , and at least relatively easy to follow the behaviour of the program in its execution .

This assertion is no more true in a parallel environment .

The tests are sometimes very difficult to implement , because it is not possible to know at a moment , what is the state of the program , implying that the interpretation of the partial results is not a simple problem in the parallel sections .

For example , in the PARFOR environment , it is not yet possible in the current implementation , to specify that a section of a program is uninterruptable or indivizable . So , if we try to print results on the screen in a parallel section , it is printed by all the processes , if they execute the same code . So , the results are very difficult to interpreat .

Another type of problem appearing only in parallel applications , is the case where the program which is to be adaptable to the available number of processes , has a different behaviour depending on the number of processes that execute it .

In one example , I was trying to debug a parallel program which was designed so that it was able to run with a variable number of processes . But the problem was the following : The program executed with one process produced normal results as they were provided by the original sequential version . Executed with 2 processes , the parallel program began its execution , and then stopped , returning immediately to the shell without giving any message nor result . The same program executed with 3 processes began its execution , and after 3 minutes , produced an error message concerning a segmentation violation . With 4 processes , the program produced the first results on a total of 26 , but wrong , and then stopped without giving any message , returning immediately to the shell . Executed with 5 processes , always the same program never produced any message , nor result . I had to break it .

This example is very signifiant to present the difficulties in parallel programming . Eventually , what was the problem in this case ? The problem was depending on the number of processes . But the program was designed completely symetric . So , the behaviour should have been the same with a variable number of processes . In fact , the error was found later . It was a problem of bad memory management in the parallel environment .

Another frequent problem is the detection of an error in a particular arrangement of the execution of a program . How could it be possible to reproduce that particular execution ? This remains a great problem in parallel programming , due to the dynamic allocation of the processes in the computer , and to the permanent unknown evolution of the environment . In fact , each execution of a program can lead to different results , due to the fact of the asynchronism , implying the difficulties when trying to debug parallel programs .

Chapter 3

Parallelization in the PARFOR and FORCE environments

3.1 Introduction

In this chapter , we discuss the possibilities provided by the PARFOR environment for parallel programming . We make also some assumptions concerning the FORCE environment for some comparisons . However , we attribute a larger importance to the programmation in the PARFOR environment , because this project is developped in our team , and it is the object of this work .

As a conclusion , at the end of this chapter , we provide some tries of comparisons between both PARFOR and FORCE environments , their philosophy , and the facilities they provide to the FORTRAN programmer .

3.2 Parallelization in the PARFOR environment

3.2.1 Introduction

In this section , we discuss the various ways a program can be parallelized with the tools provided by PARFOR . The description of the PARFOR environment itself is made in the next chapter .

The PARFOR environment can be considered as an interface between the FORTRAN programmer and the operating system running on the multiprocessor machine . The tools provided by PARFOR are designed to take full advantage of the machine facilities . Thus , large complicated applications can be parallelized within PARFOR .

Suppose that you have to parallelize a program made of some hierarchie of subroutines in the classic sequential method . Most of these subroutines are made of complicated imbricated statements and many calls to various other subroutines . Some of these subroutines are very simple and repetitive , vector operations for example ; and others are more complicated , calling themselves also other subroutines .

Such a program is very common in scientific applications . We talk about scientific applications because the FORTRAN language is more specific for these kinds of applications , but this is not restriction .

The main question , in such large applications , is to know exactly what must be parallelized , and at which level the paralleliation must occur . It is a question , because the PARFOR environment allows the parallelization at only one level of programming . So , this is not true at the first view , to know which level must be parallelized .

In this section , we introduce and explain the differences that appear between the programmations at the various levels .

But first , we give a first view to the main tools of PARFOR and the way they must be employed in a parallel program . The details are however not showed here . They are available in the next chapter describing the PARFOR environment and its implementation .

We give also some informations concerning the programmation of the barriers , and the way to divide a simple job between the parallel processes .

Then we try to clarify the relations existing between the number of processors available on the machine , the number of parallel processes initiated for a parallel program , and the number of calls to the dispatching facility .

3.2.2 View on the main tools of PARFOR

There are 3 main tools that constitute the PARFOR facilities for the programmer . These tools are called like any other FORTRAN subroutine or function .

NTASKS

The NTASKS() facility is a function that delivers to the PARFOR program , the number of child processes generated for the execution of the parallel program .

The number of processes is a standard FORTRAN integer number . The function is used in the following way :

```
npar = NTASKS()
```

where npar contains , after the execution of the function , the total number of parallel processes - 1 .

Figure 3-1 below shows a representation of the mechanism .

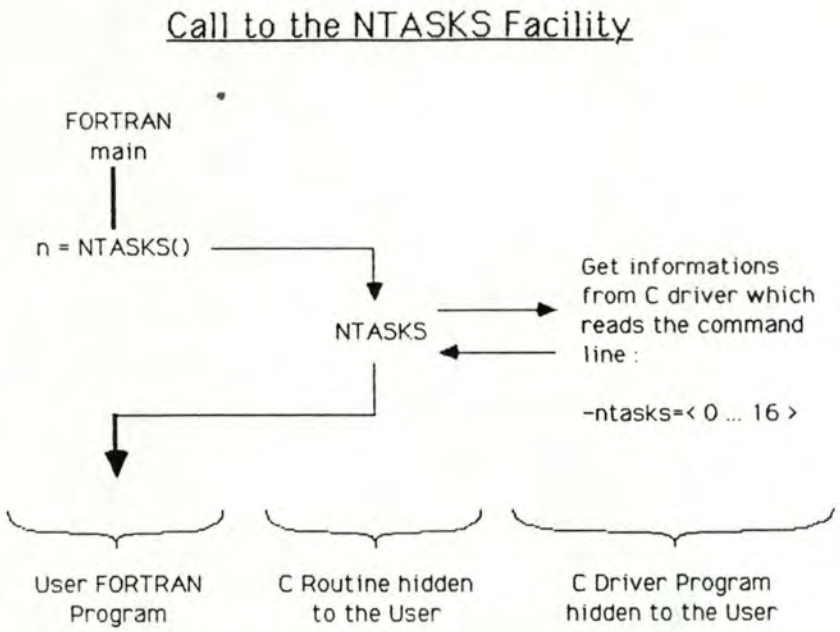


Figure 3-1

This function is important for the parallel programmer to take into account the specification of the user which executes the program . This is the only way for him to know the number of processes initiated automatically by the PARFOR environment . This can be considered as an interface between the UNIX environment and the PARFOR program .

The number of child processes is specified at the run time by the user of the parallel program . It can vary it in the range from 0 to 16 . If the user specifies 0 , the program is executed sequentially (no child process created) .

The user specifies this number by an option , like any other UNIX option in the following way :

-ntasks=<0 . . 16>

The number specified is passed to the PARFOR environment , which in turn , initiates the processes . This number can then be known by the PARFOR programmer using the NTASKS() facility that we have described .

TASKIN

The TASKIN(. . .) facility is a subroutine that attributes to an initialized parallel process , some work to do . The TASKIN facility has some parameters . These parameters are present to describe the work to do .

Figure 3-2 below shows a representation of the mechanism .

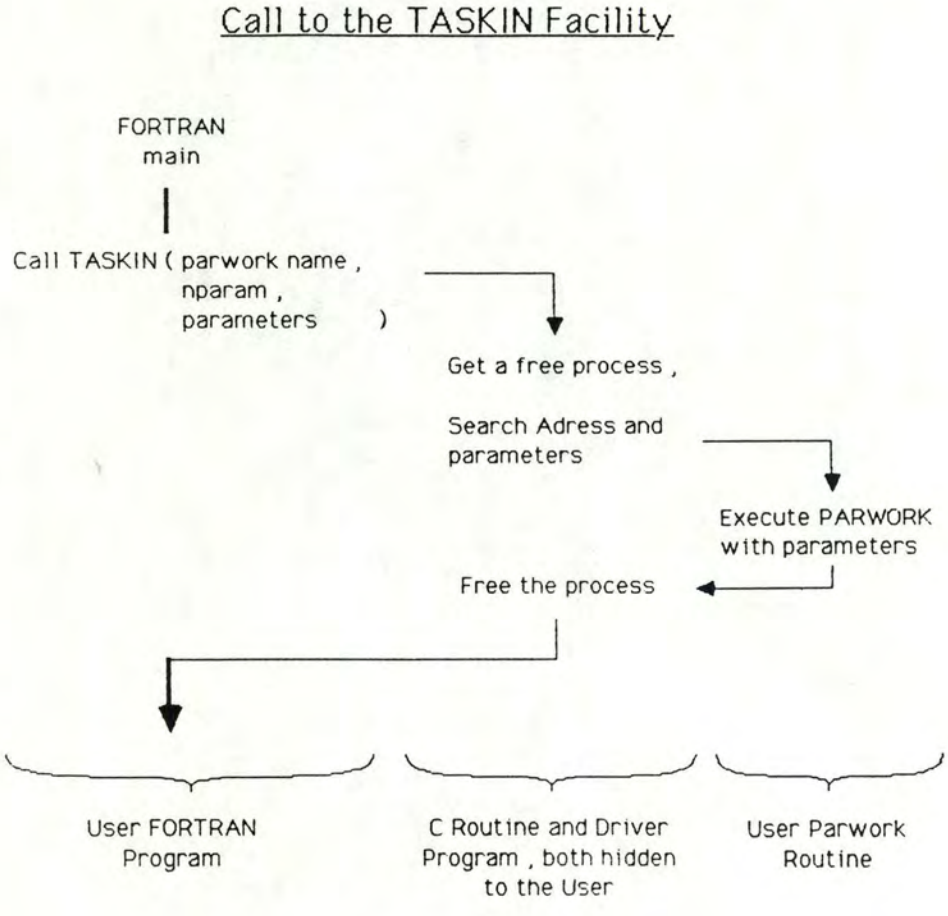


Figure 3-2

The call to the TASKIN facility is done in the following way :

```
Call TASKIN ( parwork , nparam , . . arguments . . )
```

where the parameters of TASKIN have the following meaning :

- parwork is the name of the parallel work subroutine to be executed in parallel . The parallel work subroutine must then be specified as a normal FORTRAN subroutine , bu t , taking into account that it will be executed in parallel with the others . The parallel subroutine must also be specified as external in the routine which calls the TASKIN facility . This is done in the following way :

```
external parwork
```

The reason of this external declaration is explained in the section related to the implementation of PARFOR .

- nparam is the number of parameters to pass to the parallel work subroutine . This number must be known by the PARFOR environment . It is specified in the following way :

```
data nparam /< integer number>/
```

- arguments contain the real values of the parameters that are passed to the parallel work subroutine . These arguments must be given by address . The reason is that PARFOR does not support the mechanism to pass the parameters by values . The number of parameters that are passed is equal to the nparam specified earlier in nparam .

WAIT

the WAIT() facility of PARFOR is a subroutine that makes the parallel processes to wait that all of them have finished the execution of the parallel work subroutine . Thus , WAIT() provides an easy way for synchronizations between the processes . It is a synchronization barrier . When all the processes have reached the point , the sequential code can continue until new calls to TASKIN are made .

Figure 3-3 below shows a representation of the machanism involved .

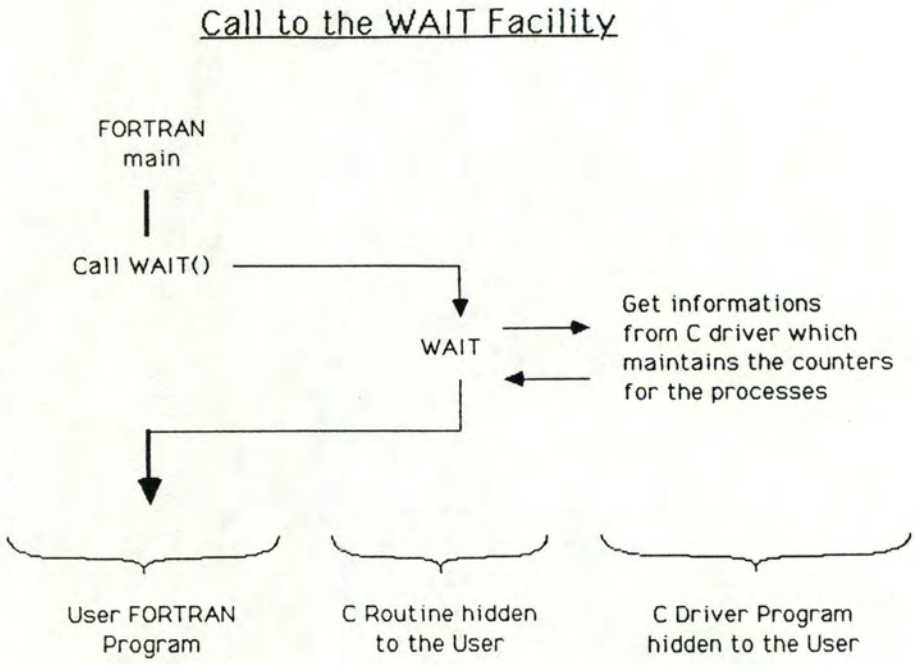


Figure 3-3

The WAIT() is called like any other FORTRAN subroutine .

```
Call WAIT()
```

Note that the implementation of the WAIT() function varies from one version of PARFOR to another . This implies a different behaviour of the processes when they are waiting , but it is transparent to the user . This is explained in the next chapter .

3.2.3 The choice of a level of parallelization

The PARFOR programmer needs to choose a level of programming for its parallelization . This is due to the restriction that is coupled with the PARFOR concept : The parallelism can only appear at one level . This restriction implies that nested parallel routines are not possible .

A single rule to parallelize , can be the parallelization of the innermost routines of a program . These innermost routines are called more often than the higher level routines . So , maybe that it is a good idea to parallelize them .

Another possible rule is the parallelization of the intermediate level routines , depending on the easiness of their parallelization . But there can be many intermediate levels of subroutines . So , the question of the choice still remains .

Another criterium for choosing the level of parallelization can be the length of the code of the subroutine to parallelize . We can parallelize the largest routines for example . But we must take care because the largest routines are not necessarily the longest in execution time . To know this , a data flow analysis would be necessary .

Another simple rule can also be the parallelization at the outermost level . This implies that the outermost routines of the program are parallelized .

Anyway , this is not a true problem to define the level of parallelization of an application . It depends on many factors , and eventually , it is always the responsability of the programmer to decide at which level he will make it , according to the granularity , according to the length of the routines in the algorithm , according to the complexity of the parallelizations .

3.2.4 The levels of parallelization in PARFOR

Initial example of a sequential program

```
Start the program
.
.
Call some complicated routines
    .
    Call some other routines
        .
        Call the innermost routines
        .
        End Innermost routines
        .
    End other routines
    .
End complicated routines
.

End of the program
```

Parallelization at the innermost level

The first level of programming with PARFOR can be the innermost level . The innermost routines of a program are parallelized . These routines are repetitive and simple . They require no synchronizations .

The main scheme of the algorithm remains sequential . The calls to the TASKIN are still sequential , the structure of the main algorithm remains unchanged , and the enclosing environment of the parallelized routines is also unmodified .

In this way of parallel programming , the standard interface of the innermost routines is preserved . The only difference in the interface is that the routines may contain some common statements necessary to record the parallel results . The calls to the TASKIN facility are made in the subroutines that are parallelized , implying that a new parallel work

subroutine must also be described (the parallel subroutine which is called by the TASKIN facility) .

The TASKIN facility of PARFOR calls a parallel subroutine that we call " parallel work subroutine " . This new subroutine is executed in parallel by the various processes .

Each process receives the actual parameters to execute the same code of the routine , but with its own values . The results of parallel work subroutine can not be given back as the standard parameters . The reason is that PARFOR does not yet support this mechanism .

The results of each process must be recorded into an array declared as shared between the processes . For example , the process number 3 will fill in the place 3 of the result array .

For this reason , each execution of one parallel process must know that it is execution number i to record the results at the correct place . This knowledge of the process number is not automatic in PARFOR . That's why a dedicated parameter must be added to the parameter list . However some parallel work subroutines do not need any space for results , if only the side effect of the routine is important , instead of the functionnal result .

Scheme of the first parallelized version

```
Start the program
    .
    < Define the common varables >
    .
    Call some complicated routines
        .
        Call some other routines
            .
            Call the innermost routines
                .
                Start loop
                    .
```

```
                    < Initialize parameters >
                    Call TASKIN ( parwork,  npar , . . .
                                  parameters . . .        )
                    .
                End loop
                .
                Direct call to parwork ( . . . parameters . . . )
                .
                Call WAIT()
                .
                Final computations
                .
            End innermost routiens
            .
        End other routines
        .
    End complicated routines
    .
End of program


Subroutine parwork ( . . . parameters . . . )
    .
    < Define common variables >
    < Define local variables >
    .
    Parallel code
    .
End subroutine
```

Parallelization at an intermediate level

This second kind of parallelization is made at a higher level . The level is an intermediate level between the innermost level and the outermost level that we describe after this section . This second level of parallelising is not very different by itself , from the first parallelization level .

The algorithm still remains mainly sequential . This implies that the parallel subroutines are still included in the sequential algorithm . No synchronizations are necessary between the processes executing the same routine on different datas .

Here , the standard interface of the parallelized subroutines is lost . The main difference between this level to program and the previous one , is the following : In the previous parallelization level , because the interface was preserved with the calling environment of the routine , only the routine itself was modified . In this parallelization at a higher level , the interface of the routine parallelized beeing lost , the calling environment must also be modified .

The subroutines to parallelize are replaced by the calculations of the boundaries for the parallel work , and a loop to call the TASKIN facility , the number of times that there are parallel child processes .

The number of times that the TASKIN facility is called , is not reduced . What is reduced , is the number of calls to subroutines . Each original call to a subroutine is replaced by a loop over calculations of the boundaries , and over calls to the TASKIN facility .

This level o programming is less transparent for the program that is parallelized . I mean that the enclosing environment is modified while the subroutine itself has dizapeared . The subroutine is replaced by the parallel work subroutine executed at the same time by the various processes . The calculations associated with the use of the TASKIN facility (boundaries , TASKIN call number , . . .) are reported to the higher level .

The problem concerning the transmission of the results to the main program remains , involving the necessity of declarations of common areas in shared memory , if the functionnal effect of the parallel work subroutine is desired .

Conclusion for this second level of programming

The main difference with the first level of programming is located in the number of intermediate calls to subroutines , which is reduced here . But this reduction does not affect in a very large way the execution time , because the extra-calls of the first level , eventually , are not very expensive compared to the more critical time consumption of the TASKIN facility itself . This second level of parallelization involves still many times calls to the TASKIN facility .

This level of programming is tested in the chapter related to the tests we made .

Scheme of the second parallelized version

```
Start the program
    .
    < Define common variables >
    .
    Call some complicared routines
        .
        Call some other routines
            .
            Start loop
                .
                < Initialize parameters >
                Call TASKIN ( parwork , npar , . . . parameters . . . )
                .
            End loop
            .
            Direct call to parwork ( . . . parameters . . . )
            .
            Call WAIT()
            .
            Final computations
            .
        End other routines
        .
```

```
        End complicated routines
        .
    End of program


    subroutine parwork ( . . . parameters . . . )
        .
        < define common variables >
        < define local variables >
        .
        Parallel code
        .
    End parwork
```

Parallelization at the outermost level

The next level of parallelization is the highest level . This level of parallelization can be considered as an extention of the PARFOR concepts . Indead , as we will see below , some new concepts are introduced in this way of programming , leading to some modifications of the original philosophy of PARFOR . However , even if this extention of PARFOR is considered , no modifications in the PARFOR interface are made . It is simply another way to consider the programmation .

In such a paralleliation level , the outermost subroutine is parallelized . The interface of the outermost level routine remain unchanged but it does not contain any more statements , except the statements for the PARFOR environment . These statements are the definition of the common variables and the initialization of the outermost routines with the TASKIN facility in a loop .

In this level of parallelising , the number of calls to TASKIN must imperatively be less than the number of parallel processes . If not , the situation will come in a deadlock . This is due to the obligation of the main process to wait the arrival of all processes at the synchronization point . If there are more TASKIN calls than the number of parallel processes ,

some processes will have to execute sequentially 2 different instanciations of the same parallel work subroutine , leading eventually to a deadlock at the synchronization point .

Only one parameter is given to the parallel work subroutine called . This parameter is the identifier number of the process . The shared memory is used . The various calculations related to the distribution of the work among the processes are made in the parallel subroutine . The entire algorithm is coded in the parallel routine with shared or private variables . The shared variables are known by all processes , and must be updated only in the squential section of a barrier , but can be readen at any time . The private variables are used for defining the boundaries , for indexes , and so on . . . This kind of parallelization uses FORTRAN barriers for the synchronizations between the processes .

This level of parallelization is very interresting because the entire algorithm is truly parallelized and executed at the same time by the various processes . This was not the case in the previous versions of the algorithm where the only parallelism was present when the TASKIN function was called .

In such a parallelization level , the synchronizations are essential . They are mainly driven by barriers , and by message exchanges , decided in the main process .

Because of the absence of locks in PARFOR , the critical sections are not possible to implement . But in most cases , a critical section can be turned to a loop executed by the main process inside the body of a barrier .

Conclusion for this third level of programming

This outermost level of programming reduces drastically the number of calls to the TASKIN facility . The synchronizations among the processes were implicit in the previous level due to the sequential main form of the program , only calling some parallel facilities at some times . It is no more the case in this level of programming , where they must be done another way . This way is essentially the use of synchronization barriers , and messages through the shared memory .

The barriers can be implemented in FORTRAN , using the shared memory like any message between processes .

The main characteristics of this way of programming is that we can really say that it is parallel programming . The most important things in a parallel program are not the calls to the TASKIN facility , but well the synchronizations between the processes . This is one of the results of our tests that we explain later .

We make some tests later to compare the various performances provided by the parallelization at the various levels .

Scheme of the third parallelized version

```
Start the program
    .
    < Define common variables >
    .
    Call some complicated routines
        .
        < Define common variables >
        .
        Initialize synchronization barriers
        .
        Do for each parallel process
            .
            Call TASKIN with the parallel routine parwork ( id )
            .
        End do loop
        .
        Direct call to the parallel routine parwork ( id )
        .
        Call WAIT() facility
        .
    End complicated routines
    .
End the program
```

```
Subroutine parwork ( id )
    .
    < Define common variables >
    < Define local variables >
    .
    Parallel code using synchronization barriers and / or
    message exchanges
    .
End subroutine parwork
```

3.2.5 Comparisons between levels 1 , 2 and 3 of programming

We have described the various levels of programming with PARFOR . Here , we make a comparison of the levels 1 and 2 , quite similar , and the level 3 of programming .

A parallel PARFOR program written in the second level of programming is essentially sequential . The parallelization is introduced by the loop calling the TASKIN facility to load the waiting parallel processes .

To the contrary , the PARFOR program written in the third level described , is essentially parallel . The parallelism is intrinsic .

The parallel program written in the second level needs for synchronizations , only to call the WAIT function , while in the third level program , the synchronizations are made by linear barriers .

These differences are sufficient to reverse completely the original philosophy of PARFOR . In the original design of PARFOR , the tools provided were proposed to allow the PARFOR programmer to modify or write a program for the inclusion of parallelism . If PARFOR is used in the third level described , it breaks through this rule .

The PARFOR programmer must initially design his program so that it is executed with many processes . It is real parallel programming , while in the previous philosophy , PARFOR was there more like an assistant to the programmer .

In terms of performances , later we make some measurements to compare both levels or levels of programming .

Figure 3-4 shows the differences in the philosophy that we have discussed above .

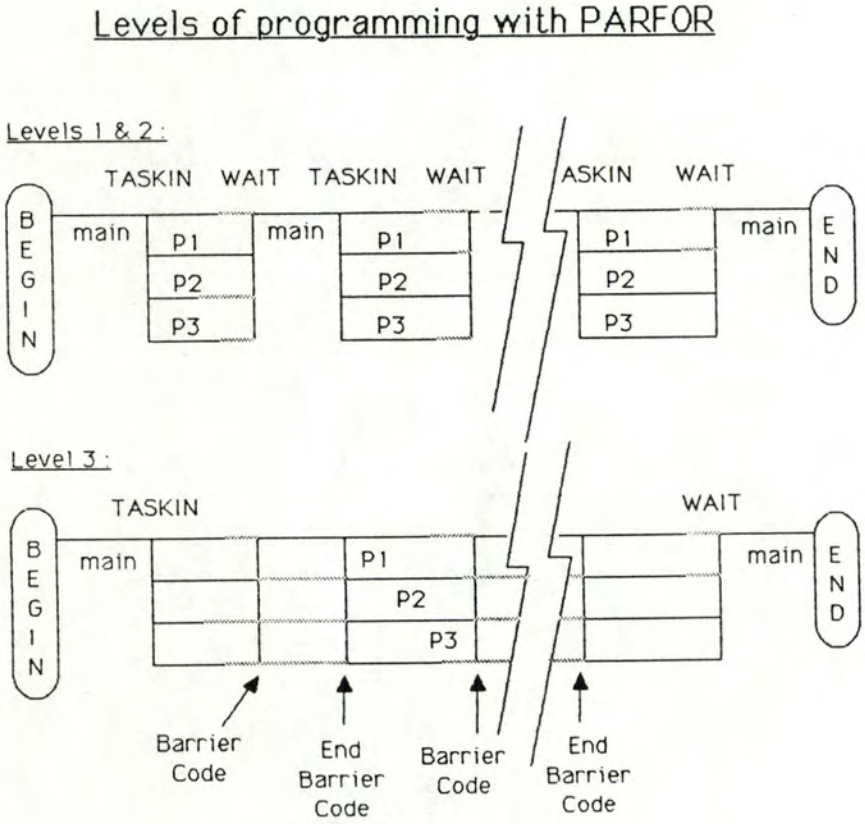


Figure 3-4

3.2.6 Examples of standard frames to program in PARFOR

Parallelization of a single loop

First , consider the case of a single loop to parallelize . The scheme of this loop is the following :

```
      Do 10 i=1 , 1000
           array[i] = calculation
10    continue
```

To be parallelized with PARFOR , an extra call to a subroutine must be added . This routine is the parallel routine executed by the various processes at the same time .

If n child processes are specified at the beginning of the execution of the program , the total number of activated processes is nchild + 1 (the main process can also do the same job in parallel with the children) .

The array of working can be divided into the total number of processes . The general scheme of the way to do that is the following :

```
C     - - - - - - - - - - -

      part = 1000 / ( ntasks + 1 )
      do 10 i=1 , tasks
           low(i) = ( i - 1 ) * part + 1
           high(i) = ( i * part )
           temp1(i) = . . .
           temp2(i) = . . .
           temp3(i) = . . .
             . . .
           tempN(i) = . . .
           call TASKIN ( PARWRK, nparam, low(i), high(i),
                         temp1(i), temp2(i), . . . tempN(i) )
10    continue
      call PARWRK ( tasks* part + 1, 1000, . . . param . . . )
      call WAIT()
```

```
C     -----------

      subroutine PARWRK ( low , high , . . . param . . . )

      integer low, high
      < declaration of the other parameters >

      do 10 i=low , high
            array[i] = calculation
10    continue
      return
      end

C     -----------
```

This kind of parallelization is called pre-scheduled . However , in this case , the number of parallel processes is determined at the run time . The algorithm is completely transparent to the number of processes . This is very interresting to have such independent algorithms , because they allow the automatic adaptation to the various configurations without any modification . But the problem in this kind of algorithm , is that it must be pre-scheduled . In fact , this algorithm does not use any shared variable . This is not dramatic , but in some cases , it would be usefull to work with self-scheduled algorithms .

This main difference between the self-scheduled and the pre-scheduled algorithms mentionned in chapter one , is that the pre-scheduled must be optimized at the compile time , so that its execution provides the best possible results . The load is equally distributed on the various processes at the run time . It involves that the execution time of each of the processes is known in advance . To the contrary , the self-scheduled algorithm allows more efficiency while the time to execute a parallel section can be unknown , or variable .The self-scheduled algorithm is based on a shared-counter . In terms of PARFOR , it could be described as follow :

```
C     - - - - - - - - - - -

      integer counter
      common /COUNT/ conuter
      . . .

      counter = 0
      do 10 j=1 , ntasks
            temp1(i) = . . .
            temp2(i) = . . .
            . . .
            call TASKIN ( PARWRK , . . . parameters . . . )
10    continue
      call PARWRK ( . . . parameters . . . )
      call WAIT()

C     - - - - - - - - - - -

      subroutine PARWRK ( . . . parameters . . . )

      < declaration section >
      integer counter , N
      common /COUNT/ counter

10    lock counter
      N = counter
      counter = counter + 1
      unlock counter
      if ( N .GT. 1000 ) go to 20
      array[N] = calculation
      go to 10
20    return
      end

C     - - - - - - - - - - - -
```

In this version , the self-scheduling is present , and there is no need to compute some boundaries of computations before the TASKIN facility is called . The most important feature is the shared variable protected by

atomic locks . These atomic locks are really the key to the self-scheduling algorithm . PARFOR does not yet support them , so , this way of programming is not yet possible .

Note that all of the shared memories between the various processes , need to be specified in the Makefile file with the option -F at the link time , and referenced in a common statement .

Use of the barriers in a PARFOR program

We have discussed about barriers . As we have said , only the software barrier can be used in PARFOR . This kind of synchronization barrier is called the linear barrier . We can define it in the following way :

```
C     - - - - - - - - - - -

C     declarations

      data            nparam /1/
      external        parwork

      parameter       ( mxproc = 30 )
      integer         lock(mxproc) , pid(mxproc)
      integer         me , i , npar

      common          /AREA1/  lock , npar , pid

C     - - - - - - - - - - -

C     Initialization of the barrier before the calls to TASKIN

      npar = NTASKS()
      do 15 i=1 , npar + 1
           lock(i) = 0
15    continue

C     - - - - - - - - - - -
```

```
C     Code of the initialization of the parallel subroutine
C     with the TASKIN facility calling the parallel work subroutine

      do 10 i=2 , npar+1
            pid(i) = i
            call TASKIN ( parwork , nparam , pid(i) )
10    continue
      call parwork(1)
      call WAIT()

C     - - - - - - - - - - - -

C     Code of the barrier executed by all processes
C     having a process number from 1 to npar + 1
C     in the parallel work subroutine .

      lock(me) = 1
      if ( me .NE. 1 ) then
10          if ( lock(me) .EQ. 1 ) go to 10
      else
40          do 20 i=1 , npar + 1
                  if ( lock(i) .EQ. 0 ) go to 40
20          continue

            < place here the eventual sequential code that is always
              executed by the driver process >

            do 50 i=1 , npar + 1
                  lock(i) = 0
            continue
      endif

C     - - - - - - - - - - - -
```

Such a code is sensible . The number of TASKIN calls should not be greater than the number of parallel processes initiated . Otherwise , the barrier will be in a deadlock , waiting forever .

The process that executes the sequential code of the barrier is known in advance , and always the same .

The flag table of the barrier must be initialized before the calls to the TASKIN facility to be sure that all the flags are set when each parallel work process start its execution .

The pid is necessary for each process to know it has the specified number , allowing such a way to filter the processes according to this vitual pid .

The code of the barrier can be repeated many times in a PARFOR program . The barrier is self-reinitialized .

Note that all of the shared memories must be specified in the Makefile file with the option -F at the link time , and the variables must be referenced in common statements . This is the way to tell the loader that the region must be created in a shared memory at the execution time .

3.2.7 Relations between the number of calls to TASKIN , the number of processes and the number of available processors

We have seen the various levels to use the PARFOR environment . In this section , we discuss the relations between the number of processors available on the machine , the total number of processes generated to execute the parallel application , and the number of calls invoked to the TASKIN facility within the PARFOR program , undependent of the number of parallel processes , between 2 WAIT synchronizations .

The most easy way to use the PARFOR facilities is to design an application so that the number of calls to the TASKIN facility is always equal to the number of parallel tasks that are initiated for this application . Then , at the run time , the user specifies that its application must be executed with the same number of processes that there are processors .

This is quite easy . However , this is not always possible . For example , a program can have a semantic constraint implying that the number of parallel processes are an even number . But the user does not know that ,

and do not need to know it . It is the responsability of the programmer to arrange himself so that the specified number of processes is always even . This implies that the application must use less calls to TASKIN than the number of processes for a section of the program .

On the other hand , it is not always quite usefull to start a number of processest equal to the number of processors on the machine . If the load of the system is high , running an application requiring the same number of processes than the number of processors , will have the effect of slowing down the application and all other users .

It may also be the case that some parallel programs need to be executed with a fixed number of calls to the TASKIN facility , or a fixed number of processes .

The criteria when designing a PARFOR program , are then the following :

1) A --> The number of processors
2) B --> The number of parallel processes
3) C --> The number of calls to the TASKIN facility

We can formalize the relations between these concepts by the equality and inequality signs :

a) = --> " Same number as "
b) > --> " Number greater than "
c) < --> " Number lower than "

According to these formulations , the possible situations which may occur are the following :

1) A = B = C
2) A = B > C
3) A = B < C
4) A > B = C
5) A > B > C

6) A > B < C
7) A < B = C
8) A < B > C
9) A < B < C

This is a no sence to compare directly the number of processors with the number of calls to TASKIN . Both concepts are only related one to the other by the intermediate of the number of parallel processes .

Situation 1 is the standard way to program within PARFOR . This case is interresting to use when the user is alone working with the operating system (closed session) . The reason is that each processor actually receives its own peace of work from the main program , in a parallel process . All TASKIN calls are satisfied by the allocation of a processor .

Situation 4 is also very usual , and occurs mainly when the user is not alone working within the UNIX environment . The number of processes is lower than the number of processors , implying that there is more chance that each process is attributed a processor at the same time . Of course , it depends on the number of current processes in the running queue , compeeling for a processor .

Situation 5 is like situation 4 . In this situation however , the parallel processes that are initiated at the begin of the application are not used (at least in the section of the program where this situation occurs) . In terms of time , this solution is expensive or not , depending on the version of PARFOR that is in use . For the version 1 , it is not too expensive bacause the waiting process is put into a sleeping state . In the other versions , where the attribution of the parallel work is done by the shared memory , this is expensive because the process is waiting in a busy loop . Eventually , this situation should be encountered only when the program contains semantic constraints that are ignored by the user who specifies an arbitrary number of parallel processes at the begin of the application . The various implementations of PARFOR are explained in the next chapter .

Situation 2 implies that one of the parallel processes is waiting doing nothing . The behaviour of this situation is the same as in the situation 5 , but the number of processes is equal to the number of processors . Such a situation should be encountered only when the user is alone on the

system , and wants to take profit of the maximum capacities of the machine . Otherwise , if some other processes from the same user , or from another user are running , some processes of the parallel application will be delayed , leading to the same cases as in situations 7 , 8 , and 9 .

Situation 3 shows a case where the number of calls to TASKIN is greater than the number of parallel processes . This situation implies that some calls to TASKIN can not be executed by a process in parallel with other processes . This , in turn , involves that the calls to TASKIN that would be delayed , are executed by the main process to avoid wasting time . When a process becomes free , it is waiting until the main process has finished its own direct call , and then receives a flag by the main , saying that it has finished its execution and prepared the new work for one parallel process . It is true that this solution is bad compared to the other solutions . But if some algorithm requires a fixed distribution of the work according to some parameter , it is possible to do that with PARFOR .

Situation 6 is similar to situation 3 except that the number of processes is lower than the number of processors . This implies that the power of the machine is not completely used . In interactive mode with other users , it is better for the speedup of the algorithm to be in this situation , otherwise , some additionnal times are consumed in waiting the availability of processors to execute the processes .

Situations 7 , 8 and 9 should never be encountered . These cases are always slower because they involve sequential execution of some parallel processes . This is just what we try to avoid with the PARFOR environment , using parallel processes to reduce the total sequential times . For the tests , we made sometimes some measurements of executions with more processes than the number of available processors . The curves related to these tests show always a wronger behaviour .

As a conclusion to this section , we think that the more critical situations are determined by the cases where

A = B : number of processes equal to number of processors
A < B : number of processes greater than number of processors
B < C : number of TASKIN calls greater than number of processes

We think that these situations should be avoided in all cases for performance reasons . They can however be employed when it is not possible to divide the parallel work in another way .

3.2.8 Various possibilities to use the PARFOR environment

The PARFOR facilities can be used in the following ways in a program :

- Call TASKIN in a loop to attribute the same work to all processes and no direct call . Then a WAIT performs the synchronization point .

- Call TASKIN in a loop to attribute the same work to all processes and the main performs the same work in a direct call . The WAIT performs the synchronization point .

- Call TASKIN sometimes to dispatch some heterogenous work to some processes . The works are executed asynchronously .

- Call TASKIN only one time per process and parallelize at a fine granularity level (= the third level of programmiation within PARFOR that has been introduced earlier) .

3.3 Comparisons between the PARFOR and the FORCE environments

3.3.1 Some comparisons

The FORCE environment , at the present time , can be considered as " the state of the art " in the FORTRAN like parallel languages in the world . This environment is well known in the scientific world working with parallel processing . The interface is standard and undependent of the machine on which it is used , and it has already been tested on many multiprocessor systems .

To the contrary , PARFOR is a new idea , and has defined its own interface . Because of this , it is not yet used and known . It must still prove its efficiency in parallel programming . However , the interface is also designed to be independent of the environment in which it is used . So , an implementation is also in a developpement state for the BS2000 SIEMENS machines .

FORCE includes a set of tools rather complete . This set of tools is particulary well fournished for synchronizations between processes . This is a critical point in parallel processing , and the problem is particulary well solved in the FORCE with the possibilities of synchronizations points , asynchron variables allowing some processes to wait implicitely when trying to read values that are not yet available , and the critical sections . All these facilities make that FORCE is very powerfull and allows the programmer to write programs with very high complexity .

PARFOR is essentially based on 3 facilities . NTASKS() , TASKIN and WAIT . These tools are somewhat poor , compared with the multiple possibilities provided by FORCE . However , they are also sufficient to allow the developpement of complex programs , taking into account that these tools can be used in various ways , as we have seen earlier . With the extentions of the concepts of PARFOR in the third way of PARFOR programming , it is

possible to reach the same complexity of algorithms , as in FORCE . The communication possibilities are the same in PARFOR and FORCE , because they use the shared central memory as the privilegied path .

On the philosophic point of vue , FORCE is designed for the programmer to take a maximum of profit of the power of the machine , taking into account that he is alone working with this machine . This implies that he uses at a maximum rate the processors for only one application at a time . All the hardware possibilities of the machine are required , expecially the hardware lock memories .

PARFOR was not designed with the same final aim . It was essentially designed to give to the programmer the possibility to introduce some parallelism in its present applications , fortunately running on a multiprocessor machine , but in cohabitation with other users .

The main scheme of a PARFOR program remains sequential , with the possibilities to execute parts of the sequential program in parallel . The main task distributes the work on the various processes (TASKIN) and the parallel children run asynchronously . Then , a synchronization point (WAIT) makes return to the sequential code . This scheme can be reproduced many times in a PARFOR program .

To the contrary , a FORCE program contains an intrinsic parallelism that is present at the early begin of the program execution . The FORCE program begins its execution with all the processes that have been initiated for him . Some parts of the program can be executed sequentially . This can be done by the use of a barrier which has the function to synchronize the processes . Then , at the end of the barrier , the program returns to a parallel execution .

With the extentions of the concepts introduced with the third level of programming , the PARFOR environment can also follow this behaviour , with nearly the same performances as FORCE provides . This is explained in a previous section concerning the parallellization in the PARFOR environment .

Figures 3-5 a , b and c below show respectively a synthetic representation of the conception of programs written at the levels 1 & 2 , at the level 3 , and in the FORCE environment .

Conceptions of programming with PARFOR and FORCE

a) PARFOR levels 1 & 2 conception :

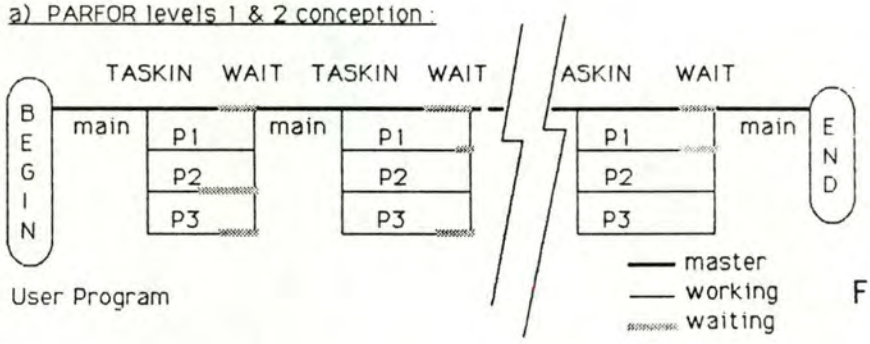


Figure 3-5a

b) PARFOR level 3 conception :

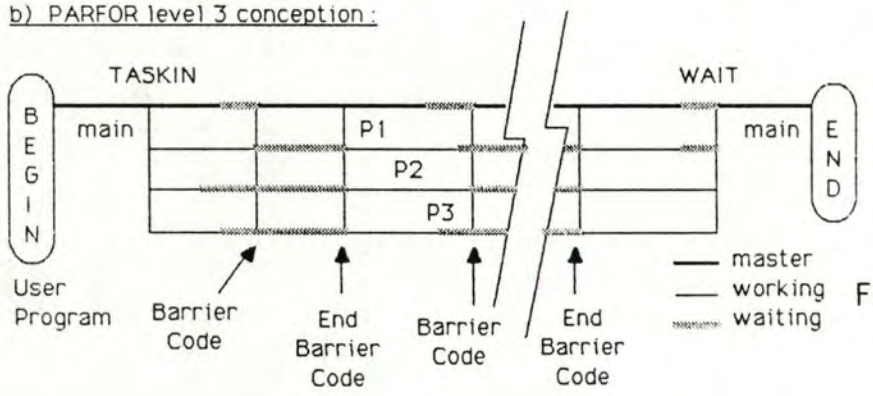


Figure 3-5b

b) FORCE conception :

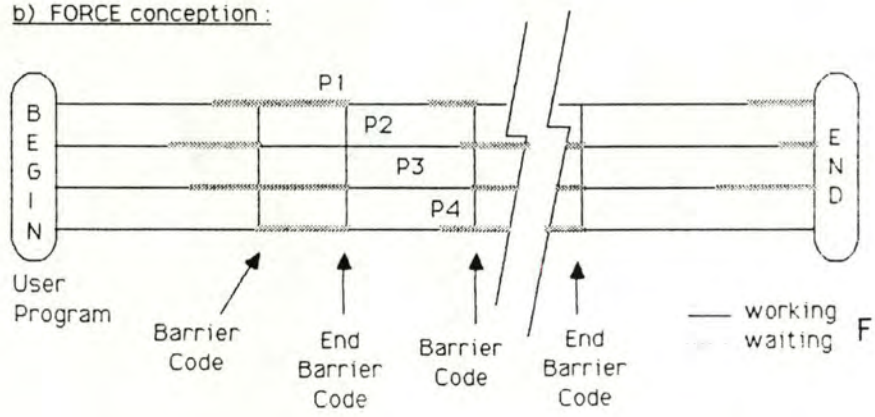


Figure 3-5c

About the granularity , the PARFOR environment can essentially treat the coarse grain and middle grain programs .

To the contrary , the FORCE environment is mostly designed for fine and middle granularity programs . This is mostly due to the possibilities of synchronizations between the processes .

On the performances point of vue , we describe in chapter 4 some results of tests made in both PARFOR and FORCE environments .

3.3.2 Personnal conclusion

To my mind , and according to the study I made of both parallel systems , they are rather similar , if we consider the extention of the concepts of the third level of programming with PARFOR .

But I thought more easy to program within PARFOR , because of the low number of concepts that are available . But FORCE was more easy in some circumstances (critical sections allowed) .

The problem of the easyness is also important , and I think that if scientific people must program in a parallel environemnt , they must be very familiar with computer programming to be able to write parallel programs with FORCE . However , the PARFOR environment is more easy because it always includes the concept of sequentiality , which is still common in most actual algorithms .

I have the feeling that the PARFOR environment is better adapted to modify an existing application , while the FORCE environment would be better to create new parallel programs .

The performances of programs written in both environments must be exactly the same . If it is not the case at the present time , it is due to the only fact that the hardware locks are not used in PARFOR . But it would be easy to allow these locks to be used .

3.3.3 Some possible extentions of PARFOR

As we have seen , the PARFOR environment could still be better than it is now . Some features could be added , as well in the implementation as in the interface with the user .

The third way of programming that we have explained could be simplified by the definition of some additionnal facilities that the programmer could call , without having to write very often the same peace of code .

It would also be possible to define some preprocessor allowing more easely future extentions in the interface , and allowing the automatization of the construction of the final parallel program .

Some little things could also be better , like the automatic availability of the virtual process identifier in the range of from 1 to < number of processes > , the suppression of the necessity to specify the number of parameters of the subroutine to call , etc . . .

The availability of the hardware locks would also be very interresting for the parallel programs . This would allow the possibilities to define critical sections , and 2-locks barriers .

Chapter 4

Implementation of parallel FORTRAN like languages on UNIX

4.1 Introduction

In this chapter , we describe how the parallel PARFOR and FORCE languages are implemented for working on a multiprocessor machine . The main characteristics of the machine itself are described in the next chapter . The operating system is the same for both PARFOR and FORCE environments . It is the standard UNIX system V provided by ATT , including the IPC libraries for communication facilities .

The first section provides a description of the PARFOR system , developped in our team at SIEMENS Munich ; and the second section is a description of the FORCE , developped by professor Jordan and his team in USA , and that we adapted for the MX500 machine .

4.2 The PARFOR implementation on UNIX

4.2.1 Introduction

In this section , we describe the main principles of working of the PARFOR environment .

The facilities provided by PARFOR are essentially a set of routines that a PARFOR programmer can use in its traditionnal FORTRAN program to parallelize some parts of its application . These facilities are only 3 in the current implementation . This small number may appear very low , but we have seen that it is sufficient to run applications having as well a coarse granularity as a fine granularity .

The parallelism treated in this environment , is processed at the process level .

A subroutine of a PARFOR program can be executed synchronously or asynchronously , depending on the way it is called in the main program .

A synchron subroutine is called by a standard FORTRAN call in the main program . Such a routine is necessarily a routine of the main program .

A synchronous subroutine is called by the intermediate of a special routine designed to prepare everything it will need before its execution . Such an asynchonous routine do not need anything to run , except its parameters and / or the common memory containing the common variables .

The asynchronous routines are said to be asynchron because they can be executed independently from the other routines independently from the main process .

The synchronous routines are said to be synchron because they are always executed by in the main process .

4.2.2 The various parts of PARFOR

The implementation of PARFOR on the MX500 machine is relatively simple . It is essentially made of 3 parts .

The first part is a driver program written is C language . This part of the environment is always the first part executed when a PARFOR program is called for execution . It can be considered as an interface between the UNIX environment and the FORTRAN system for parallel programming . His function is to initialize the FORTRAN system , reserve some shared memories , fork the processes and begin the real execution of the program to execute . Then it manages the processes and the attribution of the works . This part of the environment also contains the description of the tools available for the PARFOR programmer . It is called the cmparex part .

The second part is the FORTRAN user program including the calls to the tools of PARFOR . The program is thus a parallel program , taking into account these facilities . It is is considered by the first part of the environment , as a subroutine to execute .

The third part of the environment is a standard UNIX Makefile file in which the various parts necessary for the compilations and the links are recorded . This part is not essential , but provides a great facility when having to create , modify and debug a PARFOR program .

The environment is , as we see , relatively small . Some parts of it are fixed , and some of them are variable .

The first part of the environment is fixed . The tools and the driver program are compiled only one time . Only the object file is necessary for the linker to prepare the entire program .

The second part is variable . It is the application of the user .

The third part is the Makefile file . It must also be adapted by the programmer according to the necessities of the application . We will describe later how to built this file .

4.2.3 The tools provided by PARFOR

There are 3 main tools that constitute the PARFOR facilities for the PARFOR programmer . These tools are called like any other FORTRAN subroutine or function .

NTASKS

The NTASKS() facility is a function that delivers to the PARFOR program , the number of child processes that have been generated for the execution of the parallel program .

The number of processes is a standard FORTRAN integer number . The function is used in the following way :

```
npar = NTASKS()
```

where npar contains , after the execution of the function , the total number of parallel processes - 1 .

This function is important for the parallel programmer to take into account the specification of the user which executes the program . This is the only way for him to know the number of processes that have been initiated automatically by the PARFOR environment . This can be considered as an interface between the UNIX environment and the PARFOR program .

The number of child processes is specified at the run time by the user of the parallel environment . It can vary in the range from 0 to 16 . If the user specifies 0 , the program is executed sequentially (no child process created) .

The user specifies this number by an option , like any other UNIX option in the following way :

```
-ntasks=<0 . . 16>
```

The number specified is passed to the PARFOR environment , which in

turn , initiates the processes . This number can then be known by the PARFOR programmer using the NTASKS() facility that we have described .

TASKIN

The TASKIN(. . .) facility is a subroutine that attributes to an initialized parallel process , some work to do . The TASKIN facility has some parameters . These parameters are there to describe the work to do .

The call to the TASKIN subroutine is done in the following way :

```
Call TASKIN ( parwork , nparam , . . arguments . . )
```

where the parameters of TASKIN have the following meaning :

- parwork is the name of the parallel work subroutine to be executed in parallel . The parallel work subroutine must then be specified as a normal FORTRAN subroutine , but , taking into account that it will be executed in parallel with the others . The parallel subroutine must also be specified as external in the routine which calls the TASKIN facility . This is done in the following way :

```
external parwork
```

The reason of this external declaration is explained in the section related to the implementation of PARFOR .

- nparam is the number of parameters to pass to the parallel work subroutine . This number must be known by the PARFOR environment . It is specified in the following way :

```
data nparam /< integer number>/
```

- arguments contain the real values of the parameters that are passed to the parallel work subroutine . These arguments must be given by address . The reason is that PARFOR does not support the mechanism to pass the parameters by values . The number of parameters passed is equal to the nparam specified earlier in nparam .

WAIT

the WAIT() facility of PARFOR is a subroutine that makes the parallel processes to wait that all of them have finished the execution of the parallel work subroutine . Thus , WAIT() provides an easy way for synchronizations between the processes . It is a synchronization barrier . When all the processes have reached the point , the sequential code can continue until new calls to TASKIN are made .

The WAIT() is called like any other FORTRAN subroutine :

```
Call WAIT()
```

Note that the implementation of the WAIT() function varies from one version of PARFOR to another . This implies a different behaviour of the processes when they are waiting , but it is transparent to the user .

4.2.4 The main principle of an execution

Introduction

We can describe the main principle of PARFOR by an algorithm for each of the actors of the parallel environment . This involves the main driver program , the user program , and each of the facilities provided to the programmer . Here , we give these algorithms .

For the driver program

- Start the driver program with internal declarations
- Take and treat the flags of the command line , -ntasks option included
- Initiate the FORTRAN environment interface
- Get shared memory region for message queue
- Get shared memory region for the semaphore variables
- Get shared memory region for the arguments of the function to execute asynchronously
- Initialize protected counters

- Loop over the number of processes to create
    - Fork() the process
    - If process = the father
        - do nothing
    - If not , process = child , then do the following sequence
        - Wait for a message sent by the TASKIN facility
        - When a message present , manage the counters
        - Decode the message and find the address of memory region in which the arguments are recorded
        - Take the arguments and the address of the function
        - Give them to the assembler routine to execute it
        - Update the counters
        - Return to the waiting state for a new message
    - end if
- end loop
- Execute the subroutine MAIN = the user FORTRAN program
- Delete the shared memory regions to free the place
- End environment

For the user PARFOR program

- Start the user PARFOR program with the declarations
.
- < sequential code >
.
- Optionally call to the NTASKS facility
- Loop in FORTRAN
    - Call TASKIN facility with the subroutine name and its arguments
- End loop
- Optionally direct call to the same function
- Call the WAIT facility to synchronize the processes (barrier)
.
- < sequential code >
.
- < any other sequential or parallel code again >
.
- End PARFOR user part

For the NTASK() facility

- Start the NTASKS() facility
- Read the number of children generated , from the driver program
- End of NTASK() facility

For the TASKIN(. . .) facility

- Start the TASKIN facility with the declarations
- If the number of processes working is equal to the number of child processes
  - Give the addresses to the assembler routine to execute immediately the called parallel subroutine
  - Return to the user PARFOR program
- If not
  - Adjust the counters
  - Prepare the message text where the arguments are recorded
  - Copy the arguments in the shared memory
  - Send the message to a child that has nothing to do now
  - Return to the user PARFOR program
- End if
- End TASKIN

For the WAIT facility

- Start the WAIT facility with the declarations
- Wait that all processes have finished the execution of their subroutine , on the base of the counters .
- Return to the PARFOR user program

These algorithms are given for the first version that has been implemented . We will give some more details of the implementation in the next section .

4.2.5 Particularity of the PARFOR environment

The PARFOR environment has the particularity that at least one process is always running . This is the main process . It executes the driver , and then jumps to the PARFOR user program .

So , the driver is always executed sequentially by the main . The TASKIN(. . .) facility is then also always executed sequentially in the user PARFOR part . This is an important characteristic that we will soon explain .

In the same direction , the WAIT() facility is also always executed by the driver program . It implies that , when the driver is in the WAIT() , no TASKIN call is running .

The child processes , to the contrary , once initiated , execute always the same C code of the driver program , in a loop . Each iteration of the loop implies for the process , the execution of a new parallel work subroutine described in the user PARFOR program .

The parallelism only occurs when the parallel child processes are executing a PARFOR parallel work subroutine .

4.2.6 Location of the differences between the various versions

The differences between the versions of PARFOR are located in the mechanisms to transfer the work to the parallel work subroutine , and in the implementation of the synchronization routine WAIT() .

The various versions have been conserved to show the possibilities that are reachable for each kind of mechanism . But the main principle remains the same for all of them . Anyway , the standard PARFOR interface remains unchanged for the user .

4.2.7 Implementation of version 1 of PARFOR

Introduction

This version of PARFOR is the first version that has been implemented on the MX500 .

The particularity of this version comes from the fact that it uses messages to activate or deactivate the processes . The messages are managed by the UNIX system V IPC calls .

The procedures to manage the messages are MSGRCV to receive a message , and MSGSND to send it . The particularities of these mechanisms are that the processes are put in a sleeping state when waiting for a subroutine to be given by the TASKIN facility .

For the driver program

The detailed mechanism is the following . The driver program first , reads the number of processes choosen by the user by the intermediate of the -ntasks option on the command line . Then it initializes the FORTRAN system , and validates the signal procedure called automatically at the end of the PARFOR environment , and when a crash occurs in the user PARFOR program .

Then , 3 shared memories are reserved . The first is dedicated to the message queue . The second is reserved to the semaphore variables and has a fixed size . The third region is reserved for the arguments of the parallel work subroutine that is to be called . This last shared memory region has a size depending on the number of processes that are to be initiated . Each of these regions must be attached to a certain address of the virtual space of the process .

Protected shared counters are used for the decision of the attribution of the works to a child process or to the parent , and the same counters are also used to count the processes in the WAIT facility . The counters are the following :

- A counter of the number of processes that are busy , i.e. running , called taskbusy ,
- A counter of the number of TASKIN calls that are queued , i.e. waiting for a free process .

At this point , everything is ready for the parallel processes to be started . The loop is executed , forking the processes . The main process quits immediately the loop , while the children remain blocked , waiting for a message in the MSGRCV system V call . Each process , while waiting , is in a sleeping state , and releases his cpu .

The message , once sent , wakes up one process . The process , then decodes the content of the message .

The message contains the following informations : The address of the function to execute , its number of arguments , the number of the shared segment in which the arguments are recorded and are available for use .

From these informations , the real address of the arguments can be known , and the assembler routine can execute the function with the correct arguments . When the function is executed , the control comes back to the C part of the process .

Eventually , the counter taskbusy is updated (decremented) and the process returns to the waiting state .

For the TASKIN facility

The TASKIN facility , once invoked , contains as arguments , the address of the number of parameters of the function , the address of the function to execute , and the address of the arguments .

The first thing TASKIN performs , is a check , on the base of the present value of the shared counters , of the number of processes that are busy by a parallel work subroutine at the present time . If this number is just equal to the number of initiated processes , then , the TASKIN facility can not be executed at the present itme by the child process . It should wait that one process becomes free . Instead of this waiting , the call is

executed directly inside the TASKIN facility , which is itself always called by the driver program . So , the parallel work subroutine is executed by the driver process . When the execution is finished , the TASKIN returns immediately to the user PARFOR program .

If the processes are not all busy , then , the TASKIN call can be given to one process .

The protected counters are adapted (taskqueued incremented) , and a message is prepared with the possibility to execute the parallel work subroutine . For this , the message buffer is filled with the address of the function to execute and its number of arguments .

Then , as we know that at least one process is free , a shared memory region associated with this process is also free . The TASKIN facility searches this free region . When it is found , it records the number of this region in the message buffer .

The arguments of the subroutine are then copied in the shared memory region found .

Then , the message prepared in the buffer is sent to a child process with all informations . The TASKIN facility , eventually , returns to the user PARFOR program .

For the WAIT facility

The WAIT facility is essentially a loop that tests the values of the shared protected counters . These counters are managed so that when all the calls to TASKIN have been executed , the final state is detected . When this moment arrives , the WAIT facility returns immediately to the user PARFOR program .

4.2.8 Implementation of version 2 of PARFOR

Introduction

This version of PARFOR has been modified from the first version . It differs only in the mechanisms relative to the attribution of some work to the parallel subroutines .

The system procedures MSGRCV and MSGSND are no longer used . They are replaced by a circular table in shared memory , and 2 integer indexes to this table .

The table has an arbitrary fixed size of 100 places . The 2 indexes are called "in" and "out" and are protected by locks . Initially , the table is filled with -1 values . The main principle is the following :

For the driver program

The driver program is only modified in the various declarations for the initialization of the table , the indexes , and the suppression of the message queues and procedures . The other modifications are made in the waiting mechanism of the child processes .

Each initialized child process performs the following actions . It executes an infinite loop from which it can leave in only special conditions .

In this loop , first , it tries to get the lock2 semaphore managing the accesses to the protected indexes . When it has the lock2 , it is sure to be alone working with the values in the shared table . Then , it takes from the table , the value located at the address determined by the index "out" .

If this value is -1 , it means that this entry corresponds to no TASKIN call . Then , the child process releases the semaphore to allow the access to the other children and to TASKIN . Then , the process continues this infinite loop , waiting that the value readen becomes positive .

When this value becomes positive , it is recorded and the process jumps out of the loop . The value in the table is reinitialized at the value -1 to

say that it is free again . The "out" index is incremented circulary to the next position in the shared table , and the lock2 is released .

The shared counters managing the TASKIN and WAIT are updated , and the address of the argument is calculated on the base of the segment number taken from the shared table . Then , the address of the function and the number of parameters are taken from the shared region , and the control is given to the assembly routine that executes the subroutine .

When it is finished , the shared segment is freed , the protected counters are updated , and the process returns to the waiting state .

For the TASKIN facility

The TASKIN facility acts in the same way as in the version 1 of PARFOR , except in some small things that we describe here .

The beginning is the same as in the version 1 until the update of the protected counters . After this , the TASKIN facility searches a free region in the shared memories . A free region is marked by a value different from -1 . We are sure at this point (for similar reasons as in version 1) , that at least one region exists . When it is found , it is marked as used by a value -1 in the last place of the shared region .

The address of the subroutine to execute by the TASKIN call is copied in the first place of the shared region , the number of arguments of this subroutine is copied in the second position of the region , and then , all arguments are copied in the region .

Then The TASKIN facility tries to get the semaphore sem2 . When it has it , it is sure to be alone working with the shared table . It inserts the number of the region at the current input of the table and increases the input index . Eventually , the semaphore is released and TASKIN returns .

4.2.9 Implementation of version 3 of PARFOR

This version of PARFOR uses exactly the same mechanisms as in the version 2 previously described . The only difference resides in the way the semaphore variables are protected .

In this version , they are all protected by system V semaphores instead of atomic locks .

We do not describe here the way these semaphores are used . For more details , refer to the UNIX system V IPC reference manual .

4.2.10 Implementation of version 4 of PARFOR

Introduction

This version of PARFOR environment is not very different from the version 2 . The main difference resides in the fact that hardware locks are no more needed for the WAIT facility .

For the WAIT facility

This facility is managed by 2 counters . These counters do not need to be protected . The reason is that they are readen at any time by the children , but only updated in the TASKIN facility always executed sequentially .

4.2.11 Implementation of version 5 of PARFOR

This version of the PARFOR environment is also quite similar to the version 2 . The main difference resides in the fact that there is no more need of hardware locks for the control of the processes and for the WAIT facility .

This version of PARFOR is then implemented without any locks , taking profit of the fact that the main process is always the control process .

4.2.12 Differences of behaviour between version 1 and the other versions

The version 1 of PARFOR can be compared directly with its successors . The key difference is located in the behaviour of the child processes when waiting for a subroutine to execute .

In the original environment , the child processes , while waiting , are put in a sleeping state by the message system calls . This implies that they consume no cpu-time during this time . Sending a mesage from the TASKIN facility to a child has the effect of waking up a child process . This mechanism is automatically managed by the system calls .

In the other modified environments , it is quite different . A child process waiting for a subroutine to be attributed , remains in a busy loop , waiting for a flag to be positionned in the central memory . During the busy loop , the child reads this flag until it is set .

These various behaviours must have an influence on the effective cpu-time that a parallel application consumes while beeing executed .

In the measures , no difference should be discovered if all measures are taken in the main process .

However , there will be a great difference in the times measured in the child processes for the same application , when using version 1 or the other versions , but these differences will only appear when programming in parallel with the first or the second level of programming . We can explain the behaviour by reviewing the various possible cases :

Consider the following environments :

1) PV1 --> PARFOR version 1
2) PV2 --> PARFOR version 2 and next versions

Another parameter is the level of programming described in the previous chapter , concerning the programmation with PARFOR .

1) L1 --> First and second levels of programming
2) L3 --> Third level of programming

Eventually , the third factor affecting the behaviour of the environment , is the number of calls to the TASKIN facility .

1) HN --> A "high" number of calls to the TASKIN facility
2) LN --> A "low" number of calls to the TASKIN facility

If we combine these parameters , we have the following possible cases :

1) PV1 , L1 , HN
2) PV1 , L1 , LN
3) PV1 , L3 , HN
4) PV1 , L3 , LN
5) PV2 , L1 , HN
6) PV2 , L1 , LN
7) PV2 , L3 , HN
8) PV2 , L3 , LN

Now , we can examine the effects of these factors :

The programs where the environment PV1 is used consume always a true cpu-time . This means that the cpu-time in the parallel sections of an application is influenced in a normal way by the algorithm and the environment . For example , if a part of a program is divided into equal parallel sections , then the cpu-time consumed will be approximately similar in all these parallel sections , because the cpu are allocated to the processes only when these processes require something to do .

The programs where the environment PV2 is used , consume always more cpu-time in the child processes than required by the algorithm . Many dead times are introduced , due to the environment itself . These dead times are consumed by the parallel processes when waiting in a busy state , some work to do from the TASKIN facility . The processes are always cpu-demanding when the PARFOR environment is running .

The programs parallelized in the L1 are very influenced by the version of the environment that is used . These programs make a number of calls to

TASKIN that is relativley large . Between the calls to TASKIN , the child processes are waiting or not , depending on the version of the environment .

The programs parallelized in the L3 are not influenced by the environment used . This is due to the fact that synchronizations and all waiting times are only the problem of the algorithm . The processes are always processor demanding , undepending on the version that is used .

The programs that make many calls to TASKIN are not very influenced by the version of the environment that is used . This is due to the fact that these programs try to maximize the use of parallel subroutines , involving that the initiated child processes are very often used . The ratio occupied / free for the cpu-time of the child processes is high implying a low effect of the busy loop or not busy loop , on the cpu-time in the environment .

As a conclusion for this section , it is clear that the busy loop in the implementation of PARFOR , is not a very good solution . In terms of performances , it is however not possible to choose one version or the other without tests . In fact , the successive versions have been built to test the various possibilities provided by the UNIX system V . In chapter 5, we provide results of some tests that have been made in the various environments .

4.2.13 Compilation and execution of a PARFOR program

Compilation

A PARFOR program must first be compiled . For this , the Makefile file containing the description of all the commands must be updated . This involves the specification (according to the standard UNIX syntax of the make file) of the various parts of the application to compile , the specification of all the libraries that must be linked with the various object modules , and eventually , the specification of the shared regions names specified in the PARFOR user program .

When the Makefile file is ready , the user has just to make the file with the "make" command .

Execution

The execution of a PARFOR program is very simple . The user has only to type :

< program name > -ntasks=<number of children>

The number of child processes is equal to the total number of processes less one , for the main process .

Figure 4-1 below shows the flow of actions until the execution of a PARFOR program .

Compilation and Execution of a PARFOR Program

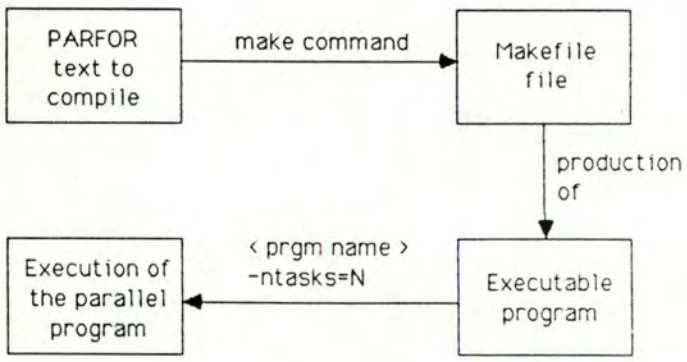


Figure 4-1

4.3 The FORCE implementation on UNIX

4.3.1 Introduction

In this section , we describe the FORCE environment in its principle of working and its implementation on UNIX .

The FORCE environment is designed for parallel programmers that have in mind to develop applications taking into account the possibilitites provided by a parallel machine . The philosophy of FORCE implies that a parallel application considers that it is alone to be executed on the machine . A maximum of processors is allocated to the application by the intermediate of parallel processes . A parallel program is executed by many processes .

4.3.2 The idea of the environment

The FORCE environment is based on a language also called FORCE , and an implementation on a particular machine .

The FORCE language is a FORTRAN 77 based language with some semantic and syntactic variations and constraints , to allow the introduction of the tools for parallel programming . Most of these differences are calls to special macros that are extended by a preprocessor before the compilation of the program can be made . So , this implies that a FORCE program is always preprocessed . A standard FORTRAN 77 compiler compiles the extended preprocessed file to produce an executable file . The language allows essentially parallel programming at both fine and middle granularity levels . Some special constructs also allow parallel programming with large granularity , allowing completely different sections of programs to be executed in parallel , without many cooperation between the processes .

4.3.3 The various parts of the FORCE

The FORCE environment implemented for the MX500 is essentially made of 4 parts .

The first part is a driver program written in FORTRAN 77 . The function of this driver program is to start the parallel processes specified by the user at the execution time , declare some shared variables that are used by all parallel processes for the synchronization facilities , initialize some procedures managing the hardware locks , initialize the barriers , and call the main part of the FORCE program when everything is ready . This is the user program . When this part of the application has finished its execution , the control returns to the driver program which in turns , kills the parallel processes and finishes the entire FORCE environment .

The second part of the environment is the FORCE user program . This FORCE part is the parallel application written with the various tools of FORCE that will be briefly described later . This part is executed in parallel by the various processes initialized in the driver program . It is considered as a subroutine to execute , for the driver program .

The third part is the shell procedure that allows the automatic construction of the final file containing the parallel program . This procedure makes calls to some small FORTRAN programs , to some script files , and eventually , to the preprocessor . It also transforms the original FORCE program to a standard FORTRAN 77 program that can be taken into account by the standard FORTRAN compiler .

The fourth essential part of FORCE is its execution procedure . This procedure allows a FORCE program to be called and executed with many processes as specified by the programmer .

The only variable part of the environment is the user FORCE program containing the parallelized application .

4.3.4 Principle of an execution of a FORCE program

Introduction

In this sub-section , we describe the main principle of a FORCE program by an algorithm for the actors that are involved at the execution time . These actors are the driver program and the user program . Here , we give these algorithms .

For the driver program

```
- Start the driver program with internal declarations
- Declare some shared regions for atomic lock memories
  shared between the processes
- Initialize the barrier variables
- Call the subroutine containing the declarations for
  shared memory regions
- Read the total number NP of processes specified by the user
  on the command line .
- Label
- If not yet forked  NP-1  processes
     - Fork the process and note the real pid
     - If I have the real pid of the father , return to label
     - Else continue
- End if
/* Note : Here is the beginning of the parallel code */
- Begin of critical section
- Execute the PMAIN subroutine = the user FORCE program
- Begin of the barrier
- End of the barrier
- If I am the father process
     - Finish the program , the environment , and kill the
       other children
- End if
- End driver program
```

For the FORCE user program

- Start the FORCE user program with many processes in parallel
.
- < User parallel code including the tools provided by the FORCE language for synchronizations and communications between the processes . >
.
- < Description of parallel and / or sequential subroutines according to the syntax of FORCE . >
.
- End the FORCE user program

4.3.5 The implementation of FORCE

Introduction

In this sub-section , we explain the way a FORCE executable program is built on the base of a FORCE source file . In the previous section , we have explained how a FORCE program is executed , involving the 2 first parts of the environment defined earlier . This section highlights the 2 remaining parts of the ennvironment , the FORCE procedure to prepare the parallel executable program , and the Forcerun procedure to start its execution .

We describe these 2 parts by an algorithm

The FORCE procedure

The FORCE procedure is the most important part in the FORCE implementation . The sequence of orders is made of UNIX commands . These commands are the following :

.

- Test the command line to check if all arguments are valid
- Preprocess all .frc files of the command line to produce object .o files
- Prepare a command line to link a load module

- Execute this command line to produce the executable version of the load module
- Prepare a temporary file containing all .o files , one on each line
- Execute the executable load module with the temporary file as data file , producing such a way a new command line .
- Execute the new command line which is a command for linking the final program . The result is the .exe final file containing the parallel program .

This procedure uses of an intermediate load module . This module is necessary for the complete automatic preparation of the final program . Its function is to prepare the command line for the linkage of the final program . The problem is that this command line is functionnally dependent on the application for the .o files to include , and the shared memory names to specify to the linker .

The load module is written in FORTRAN and makes the following :

- Begin of the load module
- Write on the standard output the first part of the command line for the linkage of the final program (name of the linker , name of the standard library and name of the standard shared area used for the management of the environment) .
- Execute the subroutine MEMSHR(INT) located in the file of the user program . This subroutine contains the declarations for the shared variables , and a write statement of the name of the shared region , on the standard output . This subroutine is automatically produced by the preprocessor in the main .f file .
- Get all arguments from the input data file and write them on the standard output
- End of the load module .

The Forcerun procedure

The Forcerun UNIX procedure is very simple . It is the following :

- **Test the command line to see if it is the correct format . If not , return to the shell with a message**
- **Read the total number of processes specified by the user and pass it to the driver program by the intermediate of a file**
- **Execute the parallel program specified , with as entry , the file containing the number of processes**
- **Delete the temporary file .**

In a further sub-section , the invocation of a FORCE program is explained .

Note about the preprocessor

As we have said , the original .frc FORCE program is first proprocessed to be transformed in a FORTRAN 77 .f file . This preprocessing is made in a multiple pipe command , with as input , the original FORCE .frc file , and as output , the FORTRAN 77 .f file .

The first 5 stages of the pipe command are subtitutions of names made by the SED editor with a script file for the description of these substitutions . The last 2 stages extend the names of the macros to their code . These extentions are made with the M4 module , a standard UNIX preprocessor , and an input file containing the semantic extentions .

4.3.6 Particularities of the FORCE environment

The FORCE environment has the particularity that when a parallel program is started , no one of the parallel processes is considered as a master process . All the processes are considered the same way implying that the system is perfectly symetric .

This involves that the peace of sequential code enclosed in a barrier construct , must be executed by only one process , but this process can be any process .

Another particularity is that a FORCE program begins immediately its execution in a parallel way . So , the parallelism is intrinsic to the environment . The program remains parallel until the join construct at the end of the program .

A FORCE program is always executed with the number of processes that has been specified by the user on the command line .

4.3.7 The tools provided by FORCE

Introduction

In this section , we will quickly review the tools available for the FORCE programmer in a parallel program . We do not describe here all the details . For more informations , please refer to the user manual of the FORCE environment .

The tools available in a PARFOR program are of various natures . But the common factor of these tools is that they are all implemented as macros that are extended to standard FORTRAN 77 by the preprocessor .

The tools for specification of the program structure

Force :

This macro declares the start of a parallel main program . It sets up the environment , and all processes begin their execution from this point until the join operation .

End Declarations :

This macro indicates to the preprocessor that the declaration part is finished .

Join :

This macro terminates the execution of a parallel program .

Forcesub :
This macro declares the start of a parallel subroutine .

Externf :
This macro informs the Force compiler / preprocessor about the necessity of external modules not included in the same file as the Force main program .

Forcecall :
This macro is used to invoke parallel subroutines that have been declared by the Forcesub macro .

The tools for the variable declarations

The variables of a FORCE program are declared with a set of macros . Essentially , there are 3 types of variables which can be local or common . The macros for the declarations are the following :

Private :
A private variable consists in a variable known only by one process . This implies that each process maintains separately its own copy of the variable , which is different from one process to another .

Shared :
A shared variable consists in a variable that is known by all parallel processes . There is only one copy of this variable .

Async :
Asynchronous variables are shared between processes . They have only one instanciation for all processes . These variables have both a value and a status . The type of the value is one of the types provided by FORTRAN . The state of the variable is always "full" or "empty" . These variables are managed only by special primitives that we review in the next sub-section .

Private Common, Shared Common, Async Common :
These variables have the same respectiv signification as described

above , but they are global variables , i.e. they are known in the entire program .

The tools for parallel execution

Pcase :
This macro allows a serie of independent sections of code to be executed by a single process .

Usect :
This macro separates multiple stream code sections of a parallel case Pcase .

Csect :
This macro begins a conditional single stream code section of a parallel case Pcase .

End Pcase :
This macro delimits the end of a Pcase construction .

Scase :
This macro is similar to the Pcase macro , but instead of a static assignment of the sections to the processes , the assignment is done at the run time (self-scheduling) .

End Scase :
This macro delimits the end of the Scase section .

Presched Do :
This macro allows the parallel execution of a loop in a prescheduled way . The work is automatically distributed in a fixed way between the processes according to the indexes that are specified .

Pre2Do :
This macro allows the parallel execution of a doubly indexed loop in a prescheduled way .

End Presched Do :
This macro delimits the end of a Presched Do construct .

Selfsched Do :
This macro allows the parallel execution of a loop as with the presched do construct , but the attribution of the work to the processes is done dynamically , in a self-scheduled way .

Self2do :
This macro allows the parallel execution of a doubly indexed loop in a selfscheduled way .

End Selfsched do :
This macro terminates the body of a selfsched do construct .

The tools for synchronization

Barrier :
This macro defines the begin of a synchronization barrier . The code of the barrier is executed by all processes . When all of them have reached this point , only one process can continue the execution , until the end of the barrier .

End Barrier :
This macro terminates a barrier . When this point is reached , the execution of the program returns to a parallel state .

Critical :
This macro defines the begin of a critical section . Only one process can enter in a critical section at a time .

End Critical :
This macro defines the end of the critical section .

Produce :
This macro allows the assignment of a value to an asynchron variable . This is the only way to assign a value to such variables . If the state of the variable is "full" , then the process that

tries to fill it , must wait until the state returns to "empty" .
Then , it marks the state of the variable as "full" .

Consume :
This macro allows an asynchron variable to be readen . If the state of the variable is "empty" , then the process trying to read must wait until it is "full" . Then it can read the variable and set its state to "empty" .

Copy :
This macro has the same effect as Consume , but the value and the state of the variable remain unchanged .

Void :
This macro marks the state of the asynchronous variable as "empty" , unconditionnally .

Isfull :
This macro returns the state of the asynchronous variable in a logical variable that can be readen by FORTRAN .

4.3.8 Compilation and execution of a FORCE program

Compilation

The preparation of a FORCE parallel program is very easy . This easyness is due to the fact that everything needed is specified by the user in only the FORCE user program . The FORCE procedure is called for the preparation of the program in the following way :

FORCE -o < exec filename > < filenames.frc >

The < exec filename > is the resulting name of the file containing the program to execute .

The < filenames.frc > are the source files , parts of the parallel application .

Execution

The execution of a FORCE program is easy . The user has only to start the Forcerun procedure with the name of the file containing the program to execute , and the total number of processes that he wants to attribute to this execution of the program .

This is done in the following way :

Forcerun < executable filename > < number of processes >

Figure 4-2 shows the process of generation and execution of a FORCE program .

Compilation and Execution of a FORCE Program

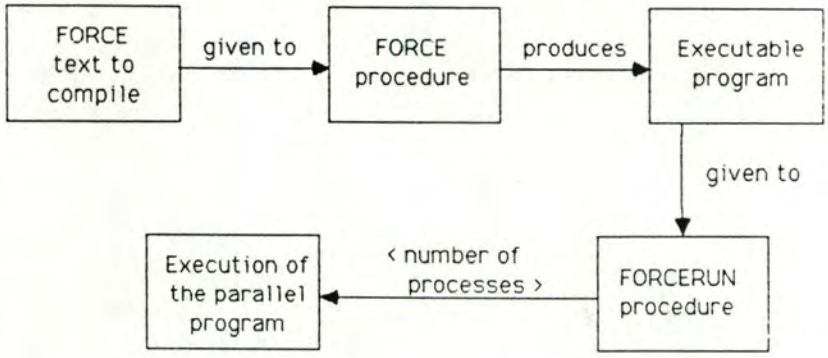


Figure 4-2

Chapter 5

Tests with the MX500

5.1 Introduction

In this chapter , we introduce the machine on which we do our tests , and the actual configuration of the machine .

Then , we report the most interresting results that we obtain , and the conditions in which they were taken , for each kind of test . We also provide a brief conclusion for each serie of tests .

Most of the tests are made for the PARFOR environment . But for the last serie of tests , we provide also results for the FORCE environment .

5.2 Summary

After many tests , we feel that the PARFOR environment is easy to use for the FORTRAN programmer . But the tools provided still suffer of youth sins . It could be possible that the PARFOR provides a better interface , without many changes to the actual design .

On the performances point of vue , we report here the results of the tests that we made with PARFOR in the ATT environment with the original version , and the next optimized versions .

The large number of results show that the ideal number of processes to execute a parallel program , is not always equal to the maximum number of available processors . They also confirm that the number of parallel processes initiated for a PARFOR program should be less or equal to the number of processors .

This number of processes is greatly dependent on the size of the parallel sections , i.e. the granularity of the program .

Our results also confirm that the original philosophy in the conception of parallel PARFOR programs , is interresting for middle and large sizes of programs . To the contrary , the tests show that , when the size of a parallel section of a program is small , the standard way of programming in PARFOR is very expensive . It is better , in this case , to program with only the initial calls to the TASKIN facility , and then , make the synchronsations inside the parallel work subroutine .

The speedup of a PARFOR parallel program depends on the algorithm , on the implementation of PARFOR , and on the level at which the program has been parallelized .

All versions of the implementation of PARFOR based on different mechanisms were tested . The original version is improved in terms of performances . The last version we wrote provides the best results for our tests . We think that this version 5 should be adopted as the standard version of PARFOR .

We also made one test within the FORCE environment . This other environment , according to the time results of this test , is also powerfull , and provides better results than those of the PARFOR environment for the same algorithm .

The PARFOR environment is , in terms of perforemances , not so good as the FORCE environment . But the difference is very little . This difference is due to the absence , in PARFOR , of hardware tools that allow the use of critical sections , as well as barriers implemented with 2 locks .

All the programs written within the FORCE environment could also be written within PARFOR . The translation from one system to the other is relatively simple . The most important modifications are located in the conversion of the critical sections into barriers . The cost of this would be a little diminution of the performances of the program compared to its FORCE version .

We must be very carefull in our assuptions , because the results that we take from the large number of tests done , are very sensitive according to the multiple parameters that we made vary . But these results provide the mean behaviour of the PARFOR environment .

5.3 The multiprocessor MX500

5.3.1 Introduction

In this section , we expose the main characteristics concerning the multiprocessor mx500 .

5.3.2 The characteristics

The MX500 system , is based on a bus architecture on which can be connected from 2 to 16 processors to constitute a full multiprocessor environment . Our test machine is configured with a set of 6 processors . The global architecture of the system is driven by a special version of the operating system UNIX V5.0 . The main characteristics of the system are the following :

- There is a high degree of coupling between the processors . All the memory is sharable among them , allowing the sharability of all resources , and communications between the processors .

- The common bus is the path used to exchange all messages between the processors .

- It is a real multiprocessor system , i.e. completely symetric . All processors can execute the system or an application at any time when they are free of work .

- All applications written for a standard one processor architecture can be executed on the MX500 system without any modification .

- The system allows dynamic load balancing , meaning that the processors use at the maximum level the possibilities of the machine . As soon as a processor becomes free , it can immediately be used for another task .

- An application can be written on the base of multiple instruction flows , all accessing to shared memory datas .

- A hardware support exists for mutual exclusion , including a set of hard locks accessible to the user .

5.3.3 General overvue of the architecture

The bus system

The main bus is a SB8000 . It is the main communication path between the processors , the memory modules and the peripherals . The rate of transmission is nearly 26.7 Mbytes in a second .

Figure 5-1 shows the global architecture of the machine .

Architecture of the MX500 machine

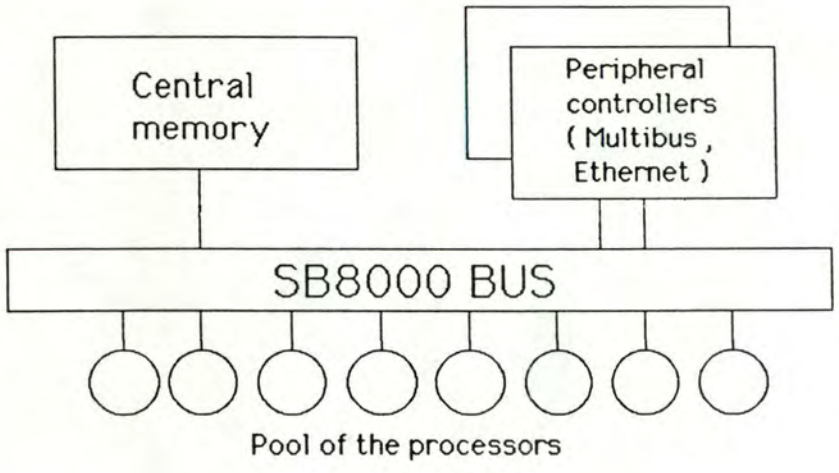


Figure 5-1

The system link and the interruption controller

A specific circuit has been developped to manage the processors connected to the system . Each of the processors has its own SLIC , as well as each board connected to the system . All SLICs are connected to a mini-bus called the SLIC-bus , which is a serial connection . These SLICs manage together the inter-processor communications , the synchronized accesses to the data structures of the operating system , the interruptions of the processors . All these operations are transparent for the final user of the system .

The pool of processors

The processors are grouped 2 by 2 on a board , but remain completely independent . Each board can be added or removed from the system . The interferences between the processors are not important , at the maximum load . Each processor is a standard NS32032 , not originally conceived to work in a parallel environment . However , the annexed circuitry takes a part of the management tasks . For example , when a processor prepares a memory access , it is the task of the annexed circuitry to validate , if necessary , a particular type of interruption . This annexed circuitry is composed by the SLIC chip managing the communications , by a bloc of 8K local RAM and 8K cache memory , by a memory management NS32032 circuit , and by a floating point NS32081 processor working in cooperation with the main NS32032 processor .

The physical architecture can support until 28 Mbytes of central memory . All this memory is available to all processors . The bloc memory allocation is done dynamically when processes recquire new space , so the use of the memory resource is nearly optimal .

The peripheral controllers

The system can be connected to a set of peripherals as disk units , tape units , terminals or others , including gates for the multibus system .

5.3.4 Performances

Performances of a single processor

The performances of one processor are nearly similar to those of a VAX750 , so that the system running with only one processor can provide the same power as the power of a VAX 750 system . When the system is configured with many processors , an equivalent number of processes can be treated in parallel , showing the advantages of the multiprocessor environment .

Addition of processors

The advantages of the addition of new processors can therefore be characterized by a better throughput of the system , and a better performance for running parallel applications , taking into account the availability of the processors .

The machine is managed by the modified version of the UNIX environment V5.0 . It is a multi-users multi-programmed operating system perfectly adapted to the MX500 multiprocessor .

The parallel applications designed to run on a multiprocessor system can take profit of the parallel architecture , and the gains in performances of these applications are influenced by the following factors :

- The percentage of the execution time of the application which must be spent in the sequential sections . According to some statistics , many applications must spent only a few time in the sequential code , i.e. 1% of the total time .

- The number of processors available in the configuration . The MX500 machine can support until 16 processors .

- The problems of contention for the accesses to the bus . But inthe MX500 machine , they are negligible , according to the constructor .

- The overheads during the creation of multiple processes . They are measured in hundreth of a second .

- The overheads due to the communications and the synchronizations between processes . They are measured in miliseconds .

5.3.5 Shared memory between processes

Shared memory regions can be declared , for several processes. Each process having an access to the shared region can easely read and write into this region . Each process can use until 8 shared regions of any size . The reservation of a shared memory region is done by a single system call (IPC system V) .

5.3.6 Applications with many instruction flows

Parallel applications can coexist with sequential applications . For example , suppose that someone starts an application requiring 4 processors . The system disposes of 6 processors . If the application has a sufficient priority , the 4 processes started by the parallel application will run , each with its own processor . Depending on the load of the system , the users can observe , at the time the application is running , an increase of the load on the system . The only 2 remaining processors can execute the code of the other users . Now , if the parallel application has not a higher priority , it will run more slowly , depending on the number of the other users working on the system . The most interresting for an application , is to have the ability to adapt itself to the number of available processors .

5.3.7 Programming languages

The MX500 , in the present days , supports the C environment , the PARFOR environment , the FORCE environment , the FORTRAN77 environment , and the ASSEMBLER language . Note that these languages do not provide by themselves the tools for parallel programming . Instead , they use the standard facilities provided with the UNIX environments .

5.4 The tools and the measures

The tools we use for providing results are described in chapter 2 in the section concerning the measurements .

5.5 Tests with the various versions of PARFOR , and with FORCE

The tests we design are made within the various environments . It means that the same programs are executed with the versions 1 , 2 , 3 , 4 and 5 of the PARFOR driver program . Some tests are not made with all versions of PARFOR , because of the unavailability of the new versions at the moment of the test , and because of lack of time , the tests were not reexecuted later with the new available versions .

Here , we remember the main characteristics of each of the versions of PARFOR . The details are available in the previous chapter describing the implementation of PARFOR .

Version 1 is the original version that uses the system messages to activate or deactivate the processes when they are called by the TASKIN facility . This version also uses some atomic lock memories for the synchronization primitive .

Version 2 is modified . Instead of using system messages , it takes profit of the shared memory , and atomic lock memories .

Version 3 uses only system V semaphores to protect the shared variables .

Version 4 uses no locks for the synchronizations , but still for the attribution of the work to the various processes .

Version 5 uses no more locks , no semaphores , but only standard C instructions .

For the FORCE environment , that's more easy because we have only one version of this environment . However , for this environment , we make only one serie of tests . The results of these tests are directly comparable with the results provided in the PARFOR environment .

5.6 The LINPACK benchmark

5.6.1 Introduction

In this section , we explain the linpack benchmark , used as a test program for the PARFOR environment .

The linpack benchmark is a FORTRAN program used for comparing the performances of various computer systems that must tackle dense systems of linear equations . The program was written by Jack J. Dongarra of the Argonne National Laboratory .

Its execution profile has a high percentage of floating point arithmetic operations . Linpack performance is measured in terms of millions of floating point operations per unit of time (megaflops) .

The program itself is based on 2 subroutines named SGEFA ans SGESL . SGEFA factors the matrix by Gaussian elimination , while SGESL solves the real system

$$A * x = b$$

using the factors computed by SGEFA .

Both subroutines call a third subroutine called SAXPY , which computes a constant times a vector , plus a vector .

In all , linpack makes 8 performances measurements and includes a consistent check in the form of a residual calculation of the results for the first of the 8 computations .

We know that linpack spends nearly 83 percent of its time in the SAXPY routine , described earlier , and 5 percent in SGEFA . A very small amount of time is spent in SGESL .

So , the most important optimization to do must be made in the SAXPY routine , by using the standard tools of PARFOR .

5.6.2 Main routines used in the main algorithm

We can describe the main scheme of the benchmark program by an algorithm . But first , the most important routines , are the following :

SECOND()

This routine is a function which gets the present time from the system and returns it in a real form .

MATGEN(. . .)

This routine fills in tables representing the linear system to solve , by a pseudo random way . Also various vectors are initialized .

SGEFA(. . .)

This routine factors a real matrix by Gaussian elimination .

SGESL(. . .)

This routine solves the real system

$$A * x = b$$

using the factors computed by SGEFA .

5.6.3 Results of the benchmark

The results of the original linpack are the following :

The first line computes various residus to check if the results provided by the calculations are correct . It means that they must be the same at each execution of the program because the same original linear system is used for each execution .

The time results of the execution are given in 2 groups of 6 columns . Each column has a special meaning .

Column 1 : SGEFA

This column provides the total time spent in the SGEFA routine which factors the system . Most of the time is spent in this routine .

Column 2 : SGESL

This column gives the total time spent in the SGESL routine which solves the system factorized by SGEFA . A little time is spent in this routine .

Column 3 : TOTAL

This column provides the total time spent to solve completely the linear system . It is the sum of columns 1 and 2 .

Column 4 : MFLOPS

This column gives the estimated power of the machine under tests in MFLOPS . This power is computed by the following way :

$$MFLOPS = \frac{OPS}{1000000 * TOTAL}$$

where

MFLOPS is the estimated power of the machine ,

OPS is the number of floating point operations necessary to solve completely the given linear system ,

TOTAL is the total time spent to solve the linear system ,

1000000 is a constant to obtain the result in MFLOPS instead of in FLOPS .

Column 5 : UNIT

This column provides the number of seconds necessary to compute a million of floating point numbers . It is computed by the following :

$$RES = \frac{2}{MFLOPS}$$

where

RES is the result described above ,

2 is a constant . The value is 2 because each floating point operation is assumed to compute 2 numbers ,

MFLOPS is the estimated power of the machine under test .

Column 6 : RATIO

This column gives the relative power of the Cray1 computer compared to the machine under test . This ratio is computed by the following way :

$$RATIO = \frac{TOTAL}{CRAY}$$

where

RATIO is the result described above ,

TOTAL is the estimated time consumed to solve the linear system ,

CRAY is the mean estimated time consumed to solve the same system . It is considered as a constant in the program .

5.6.4 Example of the presentation of the results

```
NORM. RESID          RESID                MACHEP
3.97839260E+00       7.59065151E-04       9.53674316E-07

X(1)                 X(N)
9.99704897E-01       9.99731898E-01

TIMES ARE REPORTED FOR MATRICES OF ORDER  100
SGEFA     SGESL      TOTAL      MFLOPS     UNIT       RATIO

TIMES FOR ARRAY WITH LEADING DIMENSION OF 201
3.997E-01 1.670E-02 4.164E-01 1.649E+00 1.213E+00 7.436E+00
3.998E-01 1.670E-02 4.165E-01 1.649E+00 1.213E+00 7.438E+00
4.128E-01 1.690E-02 4.297E-01 1.596E+00 1.252E+00 7.673E+00
4.091E-01 1.627E-02 4.254E-01 1.614E+00 1.239E+00 7.596E+00
```

```
TIMES FOR ARRAY WITH LEADING DIMENSION OF 200
4.049E-01 1.690E-02 4.218E-01 1.628E+00 1.229E+00 7.532E+00
4.260E-01 1.770E-02 4.437E-01 1.548E+00 1.292E+00 7.923E+00
4.308E-01 1.730E-02 4.481E-01 1.532E+00 1.305E+00 8.002E+00
4.255E-01 1.644E-02 4.419E-01 1.554E+00 1.287E+00 7.892E+00
```

5.6.5 Scheme of the main program

We need to define some constants and variables before trying to understand the measures provided by the benchmark .

T1 , T2 — Temporary variables containing the instantaneous value of the time .

Time(1,1..6) — Vector of 6 elements containing the results of the measures as described in the paragraph explaining them .

TOTAL — Contains the total cpu-time used to perform the solution of the linear system .

OPS — Constant defining the number of floating point operations performed to solve completely the Gaussian system . This number is always known if we know the size of the system to solve . In our case , the size of the system is 100 . So the value of OPS is :

```
OPS = ((2 * (100 ** 3)) / 3) + (2 * (100 ** 2))
```

CRAY — Mean estimated time to solve completely the linear system on a CRAY computer .

The basic measures are taken in the following way :

```
Call MATGEN( . . . )                          Initialization
.
T1 = SECOND()                                 Time routine
.
Call SGEFA( . . . )                           factorization
.
T2 = SECOND()                                 time routine
.
Time(1,1) = T2 - T1                           first time
.                                             computation
.
T1 = SECOND()                                 time routine
.
Call SGESL ( . . . )                          solves the linear
.                                             system
.
T2 = SECOND()                                 time routine
.
Time(1,2) = T2 - T1                           second time
.                                             computation
.
TOTAL = Time(1,1) + Time(1,2)                 Total cpu-time
.                                             consumed
.
Time(1,3) = TOTAL                             total cpu-time
.
Time(1,4) = OPS / ( 1000000 * TOTAL )         MFLOPS result
.
Time(1,5) = 2 / Time(1,4)                     cpu-time for
.                                             1 MFLOPS
.
Time(1,6) = TOTAL / CRAY                      ratio for compar.
.                                             with CRAY
.                                             computer
.
print Time(1,1 . . 6)                         print results
```

These are the basic measurements . The same scheme is reproduced 4 times in the same way , so we can get a mean of the results . In all cases , the same system is solved , so that the results can be compared .

5.7 Results obtained with the linpack benchmark

5.7.1 Tests with the linpack benchmark

The linpack benchmark that we have described , is the starting point of a serie of measures on our machine MX500 .

The tests we make have as target , the measures of the speedup of the algorithm using the PARFOR environment .

The series of tests that we made , provide a relatively large number of tables . We use these automatically produced tables later for analysis and drawing curves on the behaviour of the speedup . The tables are recorded in files .

These tests are written only for the PARFOR environment .

5.7.2 Parallelization in the PARFOR environment

We only parallelize the benchmark at the lowest level . This is the level of the innermost routines . We begin by the most often called routine , which is SAXPY . Ths routine , as we have explained at the begin of our chapter , multiplies a vector by a constant , and adds the result to another vector . The length and the start-address of the vectors are parameters .

The principle of work division between the processes is the following :

- if the length of the vector is less than the number of available processes , the sequential version is performed .

- otherwise , the vector is divided into subparts , and the bounds of each subpart are calculated so that they can be given to each parallel process.

- the TASKIN facility of PARFOR is invoked with the new parameters to pass at a sub-function executing the work .

- the main process performs the same work in a direct call to the function

- the synchronization point is reached when all processes have finished their own work on their respective region of the original vector . Then the subroutine is finished .

The same kind of parallelization is made in the other routines at the same level , but the influence of these routines is far less determinent on the global results , because they are less often called .

These routines are SDOT which computes the dot product of 2 vectors , SSCAL which scales a vector by a constant , and izaMAX which finds the index of the element having the maximum absolute value in a vector .

5.7.3 Tests with the single precision

Description of the parameters for these tests

The fixed parameters for the results are the following :

Parameter 1 : Leading dimension

This parameter corresponds to the static enclosing matrix size fixed at the compile time . This dimension is the maximum size of a linear system that can be solved by the algorithm , because of the place in the central memory .

Parameter 2 : Matrix dimension

This parameter is the size of the linear system to be solved . This number must be lower than the leading dimension .

Parameter 3 : Parallel threshold

This parameter was introduced later for several tests , but is not very usefull in this serie of tests . Anyway , it defines the threshold on the number of elements to process , under which the parallel algorithm is no more used , instead , the sequential version of the algorithm is called .

Parameter 4 : Execution number

This parameter reports the number of executions of the algorithm on which the mean results provided in the tables are computed .

These parameters are valid for all result tables .

Range of the parameters for these tests

We make vary the parameters described above in the following ranges :

- Leading dimension : 201
- Matrix dimension : 25 . . 500
- Parallel threshold : 2
- Execution number : 3 or 1 , depending on the matrix dimension
- processes : 1 . . 10

Graphical results

For more ease , we draw some graphics , showing the critical results extracted from the tables . We draw various series of curves , each of them showing some particular behaviour of the parallelized FORTRAN program .

Results of the tests made with version 1 of PARFOR

Serie 1

This serie contains the figures 5-2 a , b , c & d provided on the next page .

Each graphic shows the speedup and the efficiency according to the number of processes , with all other parameters fixed . The most important fixed parameter is the size of the matrix . We have 4 graphics for matrix dimensions of 50 , 100 , 200 and 500 elements . Some intermediate basic results for other dimensions are provided in the tables , but not reported in the graphics .

When looking to these graphics , it is surprising to see that the speedup is always the best when the algorithm is executed with one process . It is lower than one . For the execution with 2 .. 10 processes , the degradation of the speedup is clear . The results expected before the tests , were an increase of the speedup according to the number of processes . In our case , it is not the case .

Concerning the efficiency of the processors , it is directly coupled to the speedup , so , the results are also very bad . The results that we expected for the efficiency , were a curve , quasi horizontal , showing a quasi constant efficiency of the processors according to the number of processes created .

If we compare the 4 graphics , we can say that they are better for greater dimensions , but still decreasing with the number of processes . We can only say that the tendency is less bad for great sizes . For the matrix dimension 500 , there is a small positive speedup for the algorithm executed with 2 processes . But this is still weak , and the general aspect of the curve is not affected by this .

As a conclusion for these results , we would say that the parallelized version of linpack provides very.bad performances . This can be attributed to the way it has been parallelized , and/or to the environment in which it is executed .

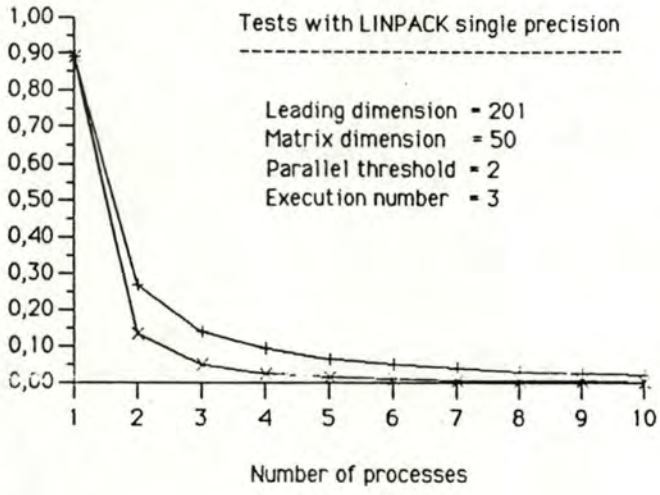


Figure 5-2a

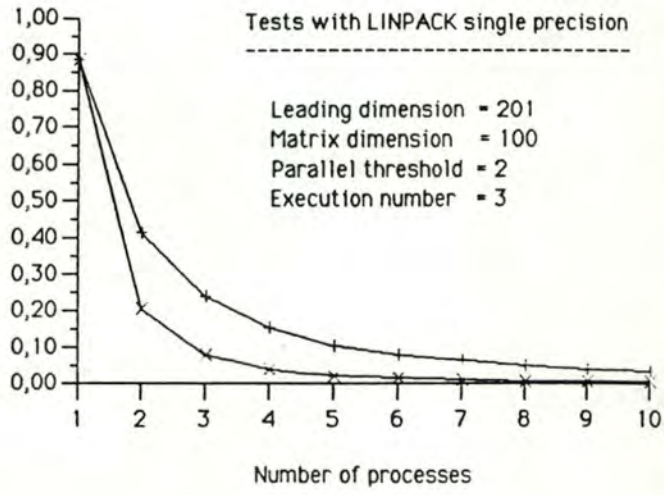


Figure 5-2b

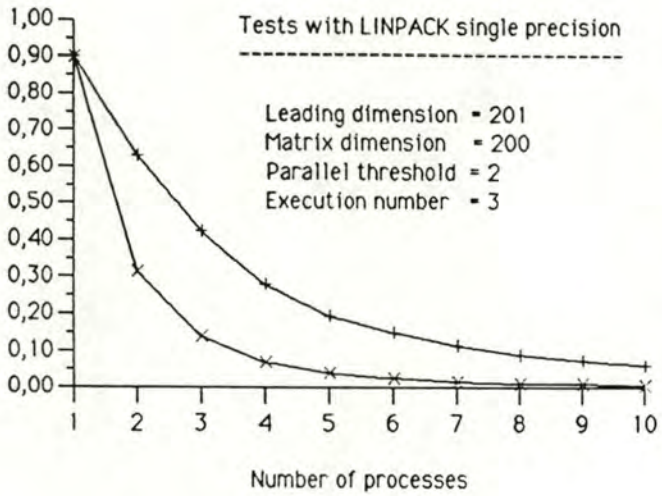


Figure 5-2c

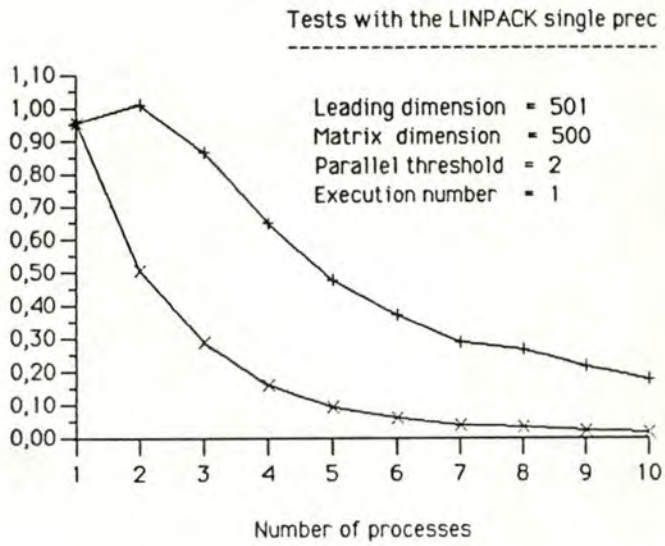


Figure 5-2d

[+] Speedup

[×] Efficiency

Serie 2

This serie contains the figures 5-3 a , b , c & d provided on the next page .

This serie of graphics shows the amount of time consumed by the parallelized linpack according to the number of processes , and the matrix dimension . The user-time , system-time and real-time are reported .

In this serie , we report 4 graphics . Each of them is the result of the execution of the algorithm with all parameters fixed , except the number of processes varying from 1 to 10 . We draw a curve for each size of the matrix .

On each graphic , the X axe shows the number of processes , and the Y axe , the time to execute the algorithm , in seconds . The absolute values of the Y axis are not very important . What we are interrested in , is the variation of the repartition of the user-time , the system-time , and the real-time .

From these 4 graphics , it appears immediately that the behaviour is nearly the same for the various matrix dimensions . So , from this , we can say that the repartition is relatively independent from the matrix dimension . We say "relatively" , because the last curve (matrix dimension = 500) shows a slight variation from the behaviour of the other curves , but weak .

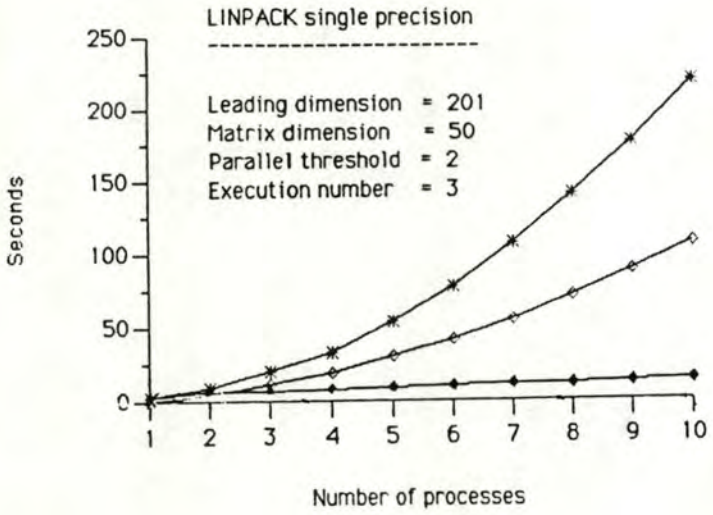


Figure 5-3a

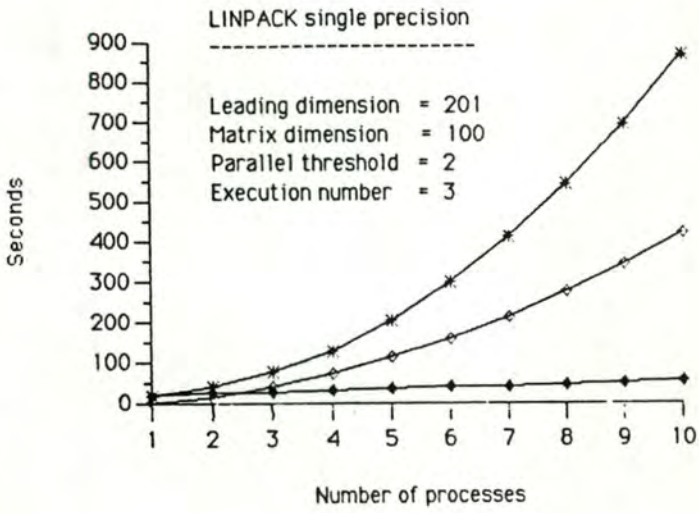


Figure 5-3b

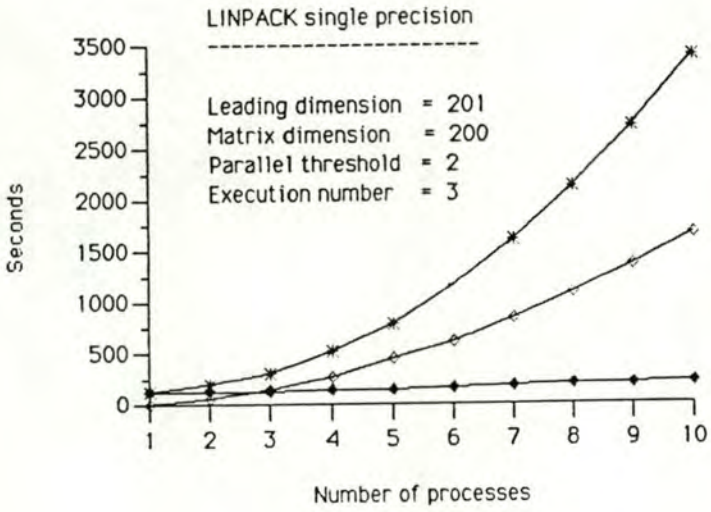


Figure 5-3c

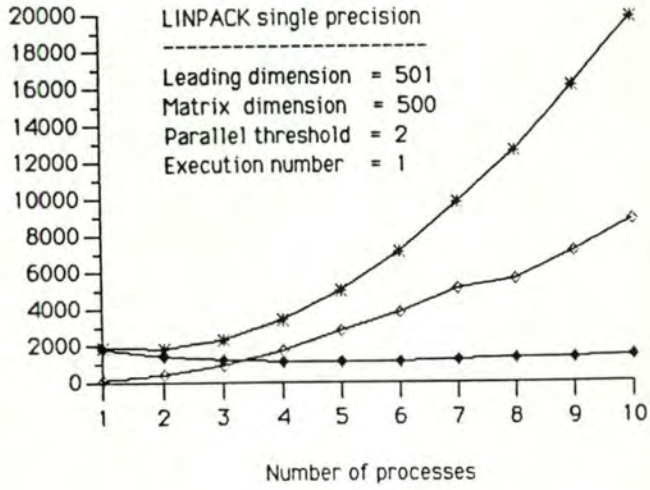
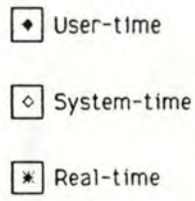


Figure 5-3d



Serie 3

This serie contains the figures 5-4 a , b , c & d provided on the next page .

This serie of graphics is the complement of the serie 2 . It shows , in terms of percentages , the repartition between the user and system times .

We reproduce the 4 graphics of the serie 2 , for matrix dimensions of 50 , 100 , 200 and 500 .

As we saw previously in the serie 2 , the repartition is nearly the same for each of the 4 graphics .

As a conclusion to the series 2 and 3 , we can see and claim that the user time follows a logical behaviour (reduced user-time according to the reduced task to achieve) , but the system-time follows a strange behaviour . More the number of processes increases , more the system time also increases , in both absolute value , and relative value to the total cpu-time .

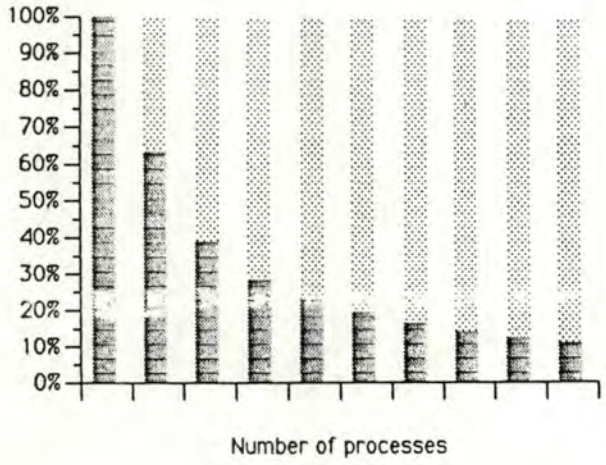


Figure 5-4a

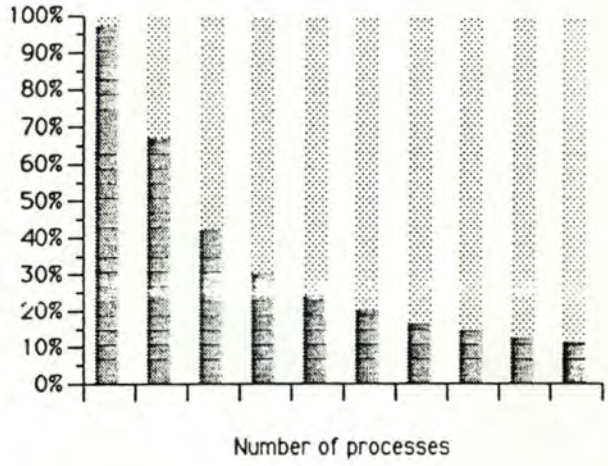


Figure 5-4b

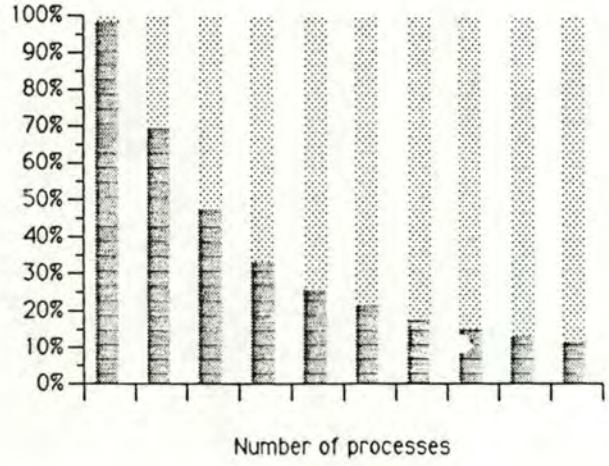


Figure 5-4c

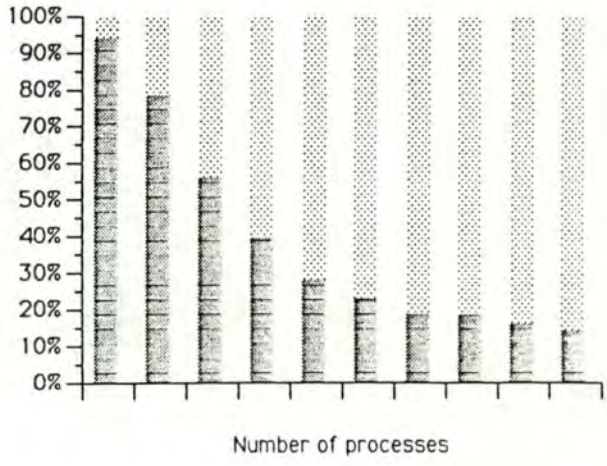
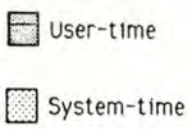


Figure 5-4d



Serie 4

This serie contains the figures 5-5 a , b , c & d provided on the next page .

This serie of graphics is intended to show in a microscopic way , the tendency of the cpu-time . the previous graphics , because of their too high scale , did not show very well this behaviour .

The serie includes 4 graphics , for matrix dimensions 50 , 100 , 200 and 500 elements . The other parameters are fixed .

We can observe that the user-time varies from a convex-like increasing curve , to a concav-like curve with a minimum .

The expected behaviour would have been a decreasing curve for the user-time , because of the diminution in terms of part of the global task to perform .

This particular behaviour of the user-time is probably due to the time consumed for the synchronizations between the processes , almost when the number of processes is relatively high and the parallel work very littl . But we can not give , at the present time , a better explanation of this strange behaviour .

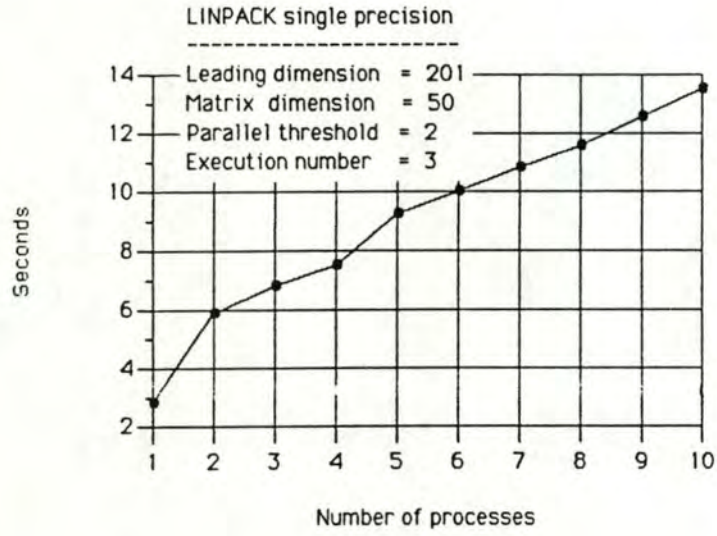


Figure 5-5a

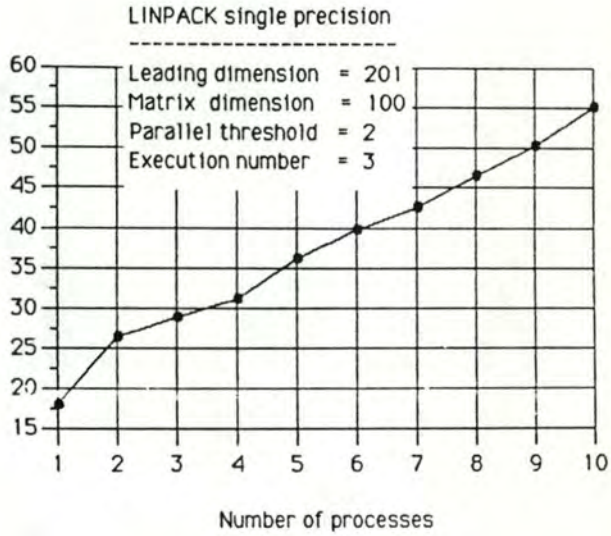


Figure 5-5b

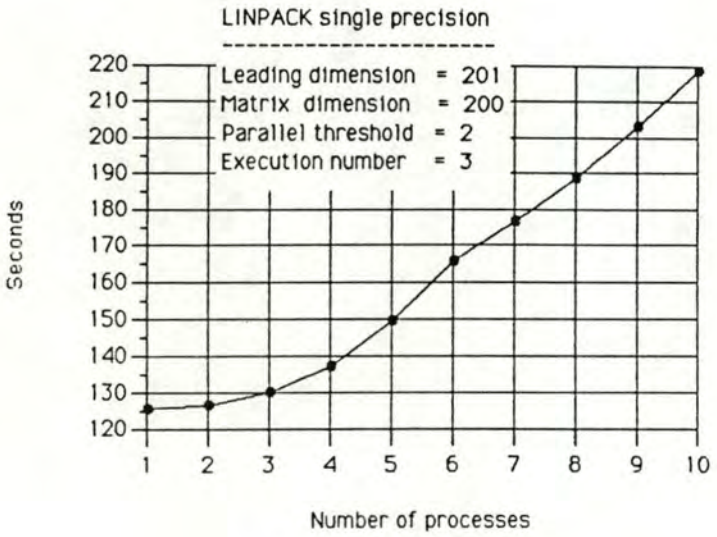


Figure 5-5c

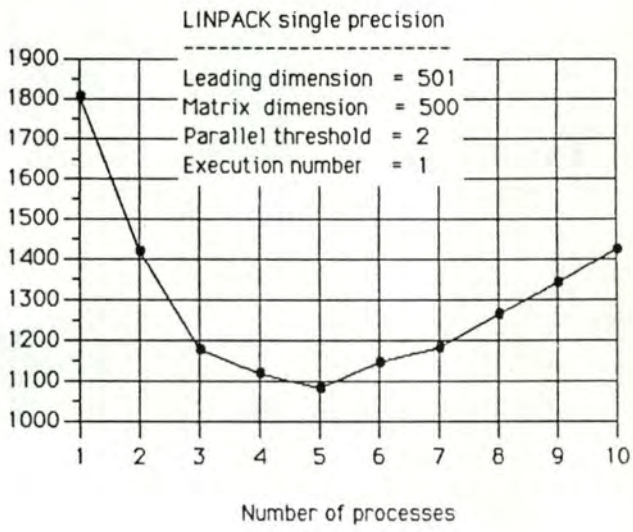


Figure 5-5d

[•] User-time

Serie 5

This serie contains the figures 5-6 a & b provided on the next page .

This serie of graphics shows only the point at which the system-time becomes greater than the user-time . We consider only matrix dimensions of 200 and 500 elements .

We can see that this point does not vary too much according to the matrix dimension .

These graphics confirm the assumption we made about the serie 2 , and serie 4 .

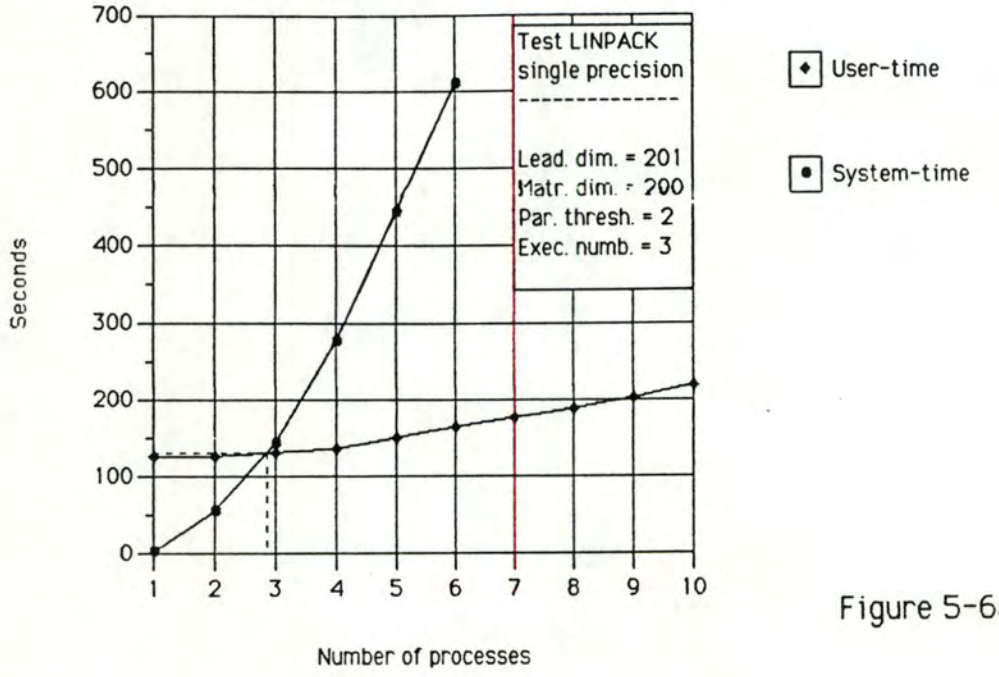


Figure 5-6a

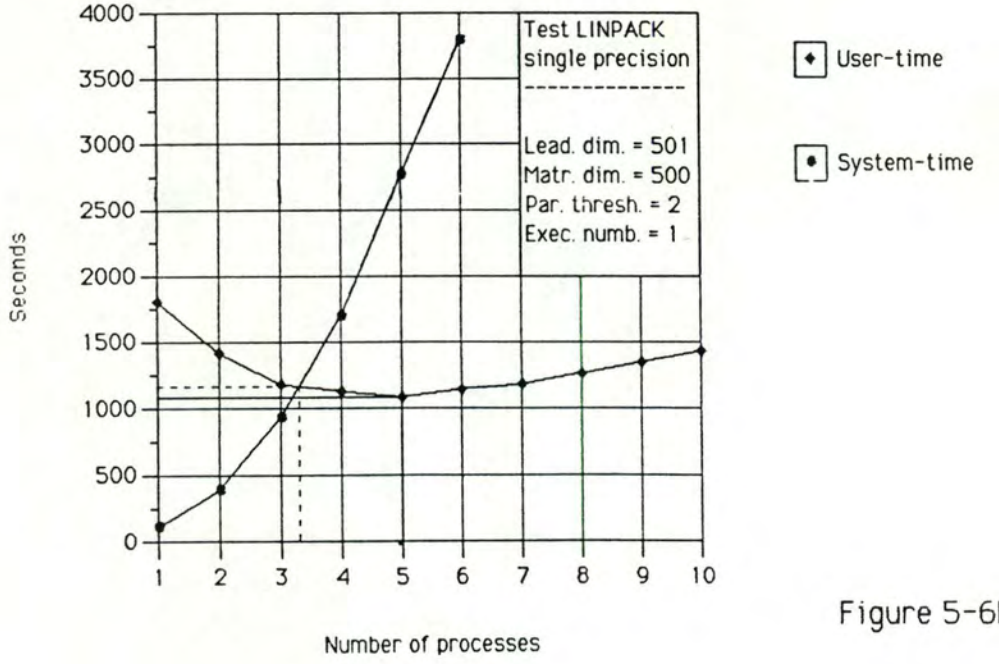


Figure 5-6b

Serie 6

This serie contains the figures 5-7 a , b , c & d provided on the next page .

This serie of graphics shows an analysis of the behaviour of the real-time compared to the cpu-time for execution of the algorithm with a variable number of processes .

For each graphic , all the parameters are fixed except the number of processes . We draw a graphic for matrix dimensions 50 , 100 , 200 and 500 .

This analysis shows immediately that the cpu-time remains always nearly half the value of the cpu-time .

We can conclude that the value of the relative repartition of the real-time , is independent from the matrix dimension . We must remember that all measures are taken in a block time environment , so , we were the only user working on the MX500 machine at the tests time .

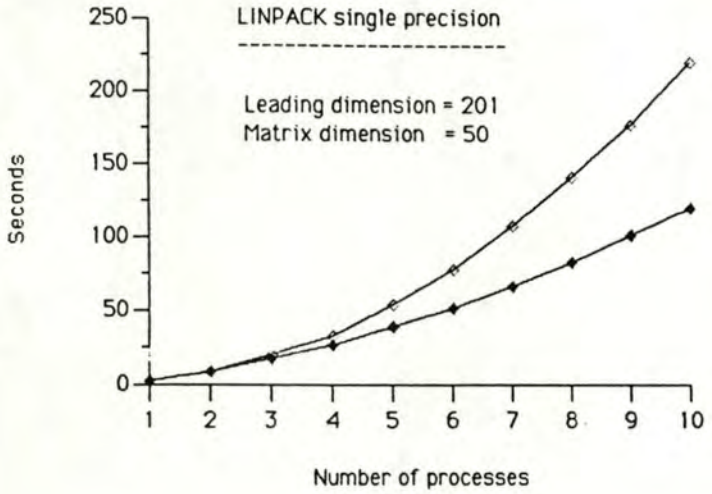


Figure 5-7a

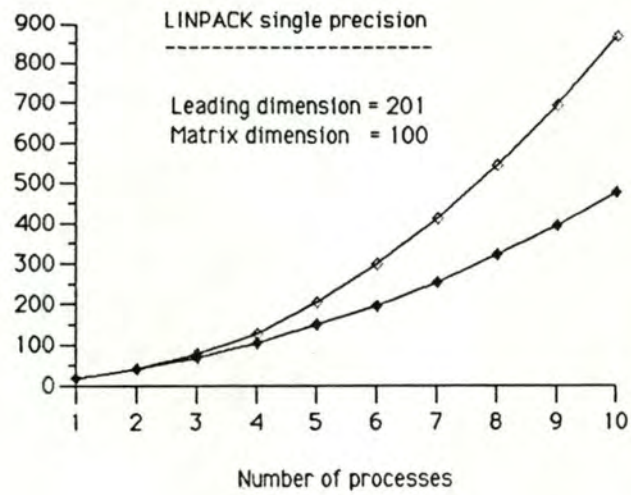


Figure 5-7b

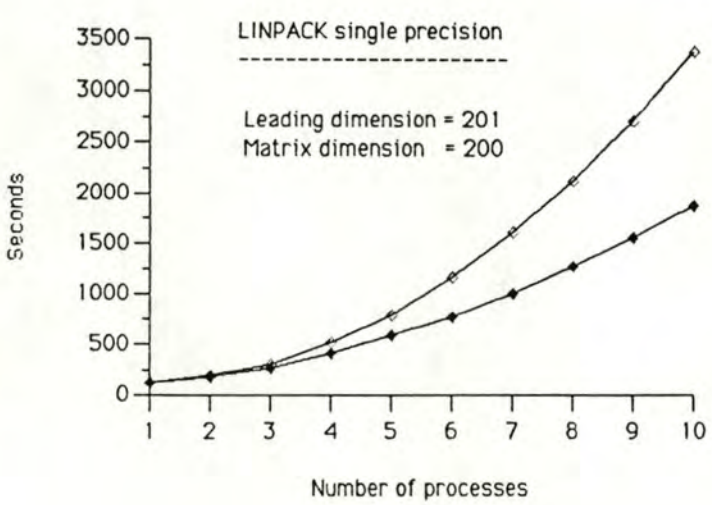


Figure 5-7c

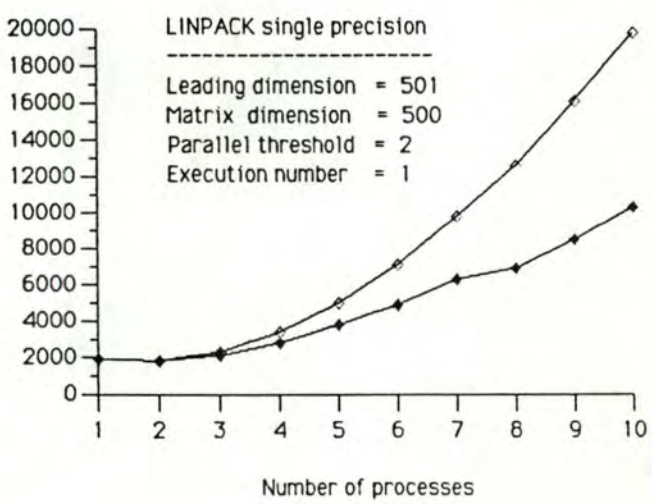
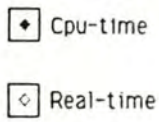


Figure 5-7d



Results of the tests made with version 2 of PARFOR

Serie 1

This serie contains the figures 5-8a to 5-8f provided on the next page .

This serie of graphics shows the evolution of the cpu-time of the entire program when the size of the matrix is increased , i.e. , when the granularity of the program is increased , and thus the size of each parallel process .

From these grphics , we can conclude that the behaviour is quite similar to the behaviour showed with the original PARFOR environment . The cpu-time for small matrix sizes is increasing with the number of processes , showing that the overheads introduced by the use of PARFOR are very large for small sizes of the parallel work subroutine . When the size of the matrix increases , we can see that a minimum appears in the curve associated with the fixed size and increasing numbers of processes .

When the cpu-time is minimum , the speed of the parallel program is maximum . The point of the minimum shows the number of processes providing the best results .

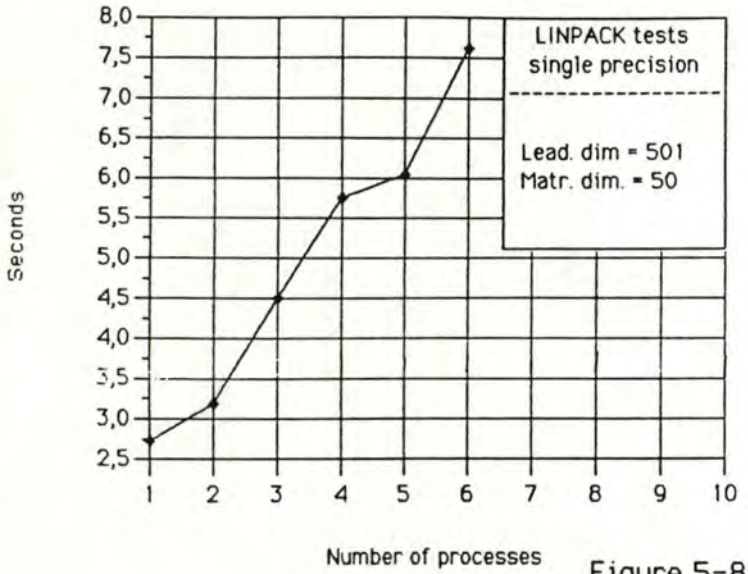


Figure 5-8a

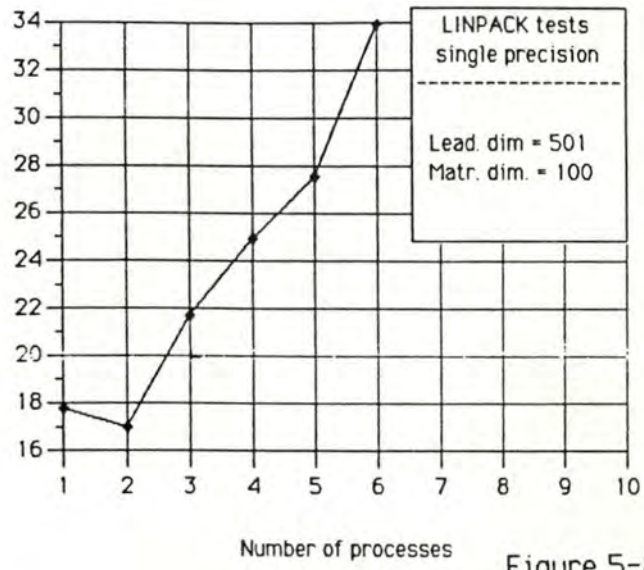


Figure 5-8b

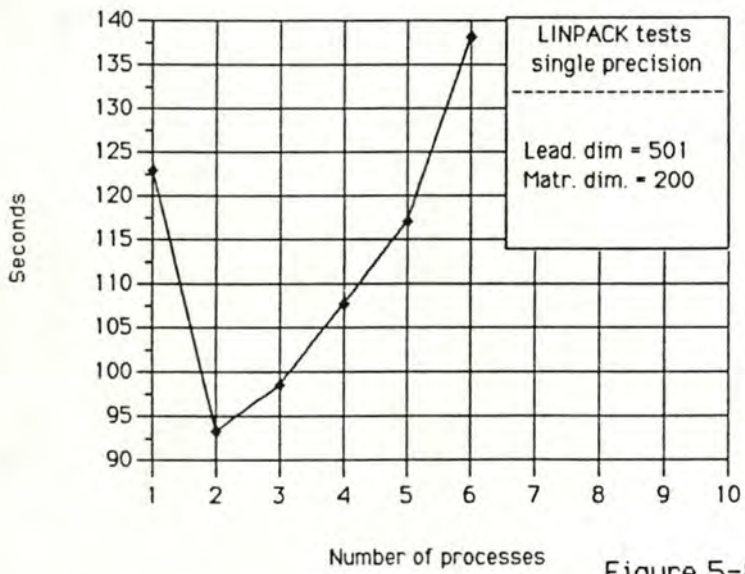


Figure 5-8c

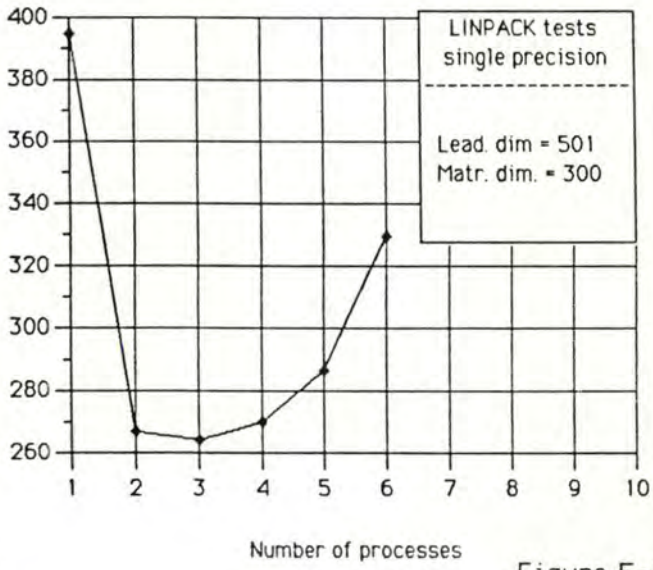


Figure 5-8d

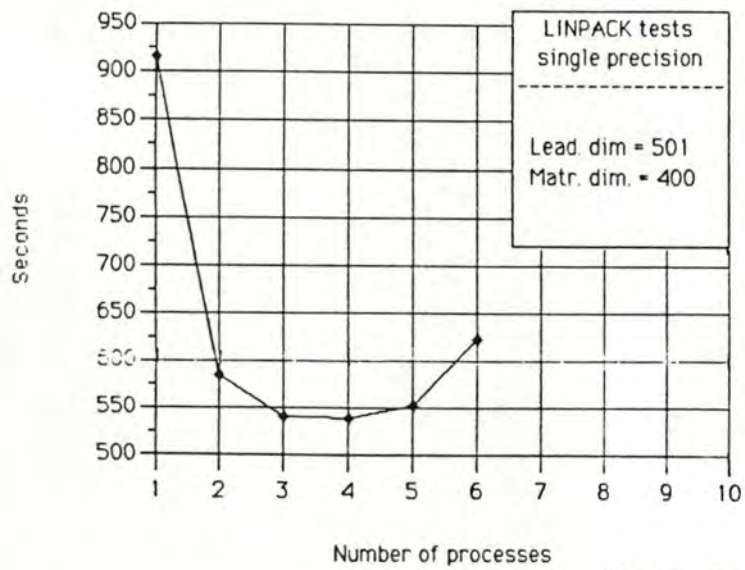


Figure 5-8e

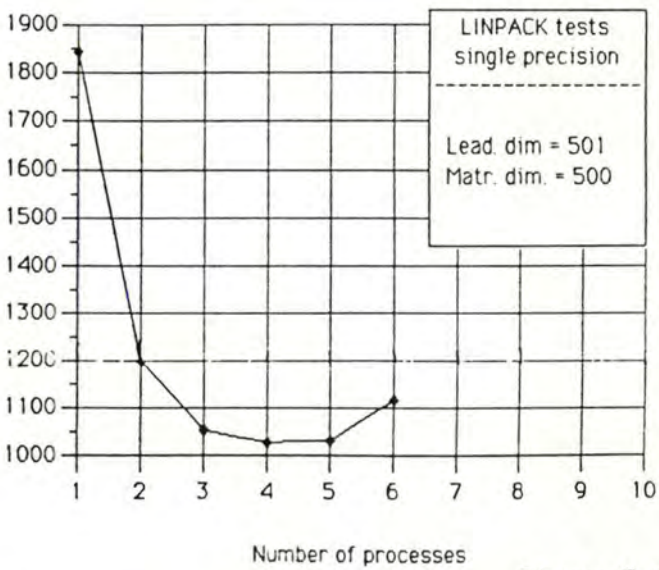


Figure 5-8f

Graphic of speedup

This graphic (figure 5-9) shows the speedup of the parallel program compared to the sequential version . A curve is drawn for various sizes . We have here the confirmation that the PARFOR is better with large parallel sizes .

The speedup is greater than one for at least 3 parallel processes when the parallel size of the matrix is greater than 100 elements . Compared to the results of linpack provided by the same program executed in the original PARFOR environment , there are more greater speedup than 1 . Remember that the speedup was only greater than one with matrix sizes greater than 400 in the original version .

We make further tests to verify this assumption , and measure the exact overheads introduced by PARFOR , with the various versions of the implementation we made .

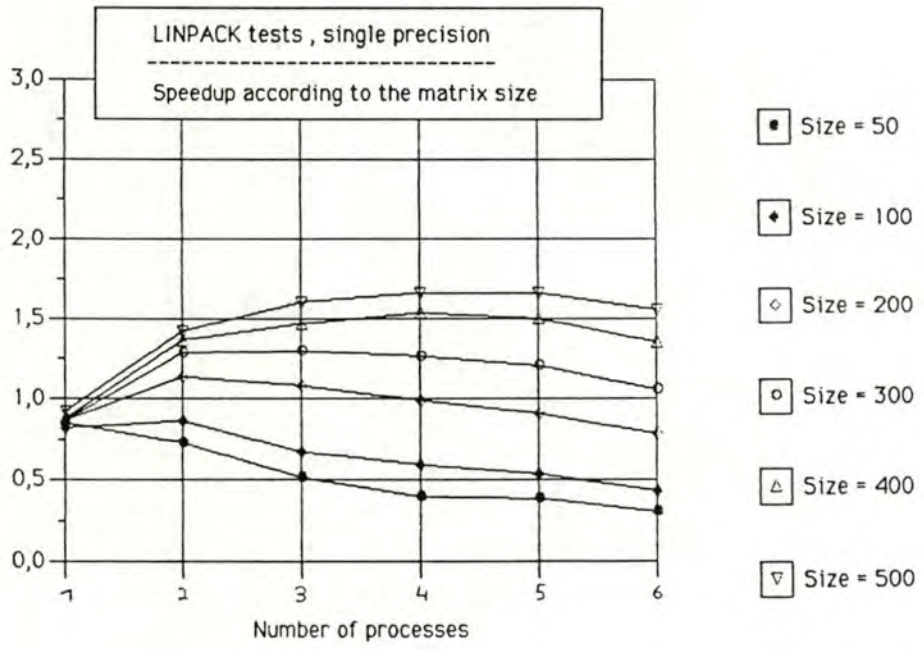


Figure 5-9

5.7.4 Tests with the double precision

Description of the result tables

The results provided by the double precision test program are exactly the same as those for single precision . The tests were done so that the results can be compared in the same conditions .

Scheme of the measures

The scheme is the same as it is for the single precision :

Time(1)

.

Execute the entire algorithm

.

Time(2)

.

All performance computations

Conclusions for the results with the double precision

We did not draw any graphic for the results with the double precision because the scheme of the graphics were exactly the same as they were for single precision .

The only variations were located in the abolute values of the real-time and the cpu-time , which are greater for the algorithm executed in the double precision .But this is not surprising , because the same behaviour was already observed in the original sequential linpack program for double precision .

5.8 Tests with the SAXPY routine

5.8.1 Introduction

At the view of the so bad results obtained with the linpack benchmark , we decided to test in a direct way the most important routine responsible for the results of linpack .

This is the SAXPY routine . In a few words , we remember that this routine multiplies a vector by a constant , and adds the resulting vector to another vector . A study made by [Johnstone] shows that most of the time of linpack is spent in this routine .

5.8.2 Description of the parameters for these tests

For these tests , we provide the results in the same kind of tables than these produced for the previous results of the linpack tests .

The parameters for the execution of the test program are the following :

Parameter 1 : Leading dimension

This parameter is fixed at the compile time . It corresponds to the maximum vector length on which the program can be executed .

Parameter 2 : Vector length

This parameter is the size of the vector on which the tests of the SAXPY routine are built . The specified size must be lower than the leading dimension .

Parameter 3 : Parallel threshold

This parameter is not used here , and is always null .

Parameter 4 : Execution number

This parameter specifies the number of executions of the algorithm . More this number is high , more the results are accurate , but more the time to execute the tests is high . So , a compromize is to be decided .

5.8.3 Description of the tests

The tests made for this routine are based on a comparable scheme as the tests for the entire linpack benchmark .

First we execute the sequential version of the SAXPY routine , then the parallelized version , but with only one process , and third , we execute the parallelized version , with the specified number of processes .

5.8.4 Scheme of the results

The scheme for taking the measures is always the same :

```
Time(1)
.
Execute many times the SAXPY routine
.
Time(2)
.
All performances computations
```

The results are generated in tables at the last phase .

5.8.5 Graphical results

Graphic 1

This graphic (on figure 5-10a) shows the measured speedup of the algorithm when it is executed with many processes .

The tests made to draw this graphic were designed with various lengths of the vector , and all other parameters were fixed . The graphic includes a curve for vector lengths 100 , 200 , 400 , 800 , 1000 , 2000 , 4000 and 8000 .

The X axis shows the number of processes , and the y axis shows the speedup in terms of cpu-time .

On the graphic , it is clear that the parallelized routine should not be used for small numbers of elements . This conclusion is very important for linpack , in which the SAXPY routine is used with vector lengths varying from <mtrix dimension> to 1 . This means that , while the standard linpack benchmark , is executed with a matrix dimension of 100 , the SAXPY routine is always called with vector lengths of less than 100 . And unfortunately , this is the badest region of working for the parallel SAXPY routine , as the graphic shows us .

The graphic shows that the SAXPY routine should not be used with many processes under a length of 400 elements . And for this vector length , not with more than 2 processes , for providing the maximum speedup . The curves with more than 400 elements show also a maximum , indicating the ideal number of processes at which the maximum speedup is reached .

We can observe that the curves for small numbers of elements , are far from the linear speedup , according to the number of processes . But more the vector length increases , more the behaviour of the algorithm becomes close to a linear speedup . The best behaviour is reached for a length of 8000 elements . We did not go further in vector lengths because the time to execute the tests is too long , and secondly because we think that these results are sufficient enough to conclude . But we can easely extrapolate the behaviour for greater lengths : A speedup closer and closer to the linearity , with an increasing number of processes that increases .

But from this behaviour , we can also think that the real problem of the bad behaviour of the parallelized version is the problem of the overheads appearing when trying to use the PARFOR facilities with relatively small tasks . In the case of the SAXPY routine , the parallel work is too small compared to the overheads generated by the PARFOR environment itself .

We think that a further analysis of the overheads of the PARFOR environment is a necessity . That's what we do in further tests .

Graphic 2

This graphic (on figure 5-10b) is the complement to the previous one . It shows the efficiency of the algorithm , in terms of cpu-time .

As for the speedup above , we draw a curve for each vector length reported . But the numerical results for other lengths are available in the appendices .

The graphic can be commented as the previous , in terms of efficiency . We can clearly observe that there is a large decrease of the efficiency of the algorithm when the vector length is very low . The efficiency is better for larger vectors , but still decreasing .

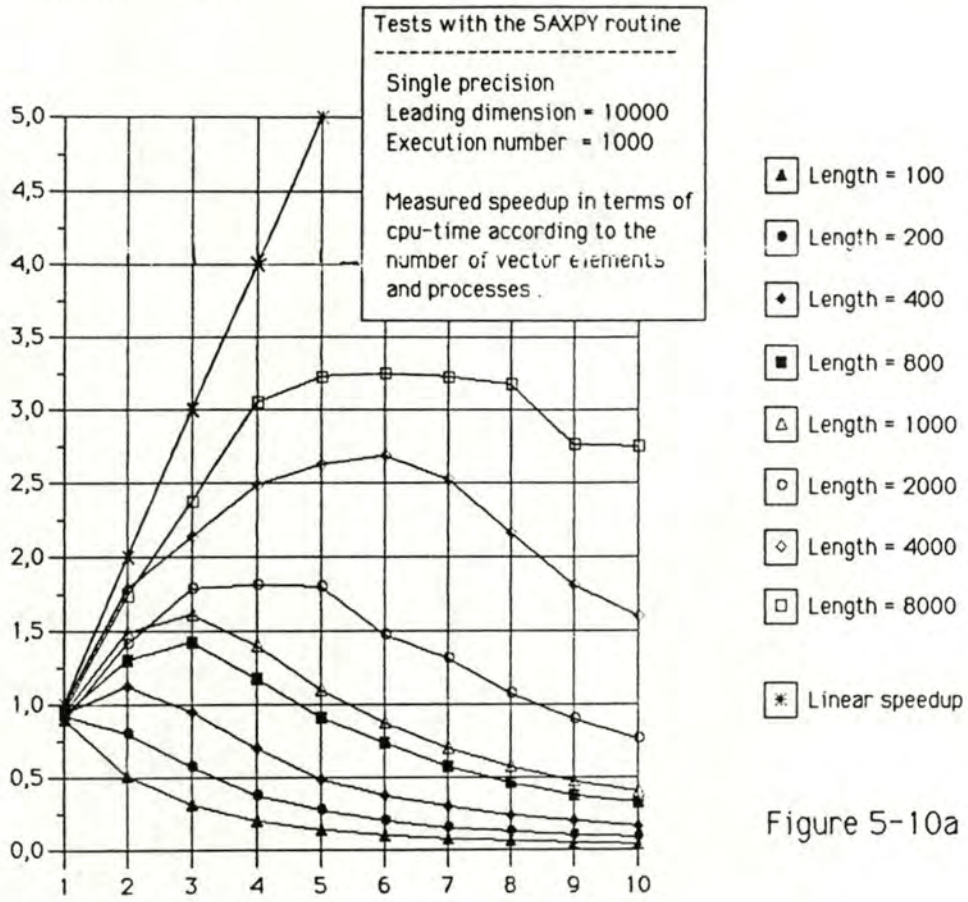


Figure 5-10a

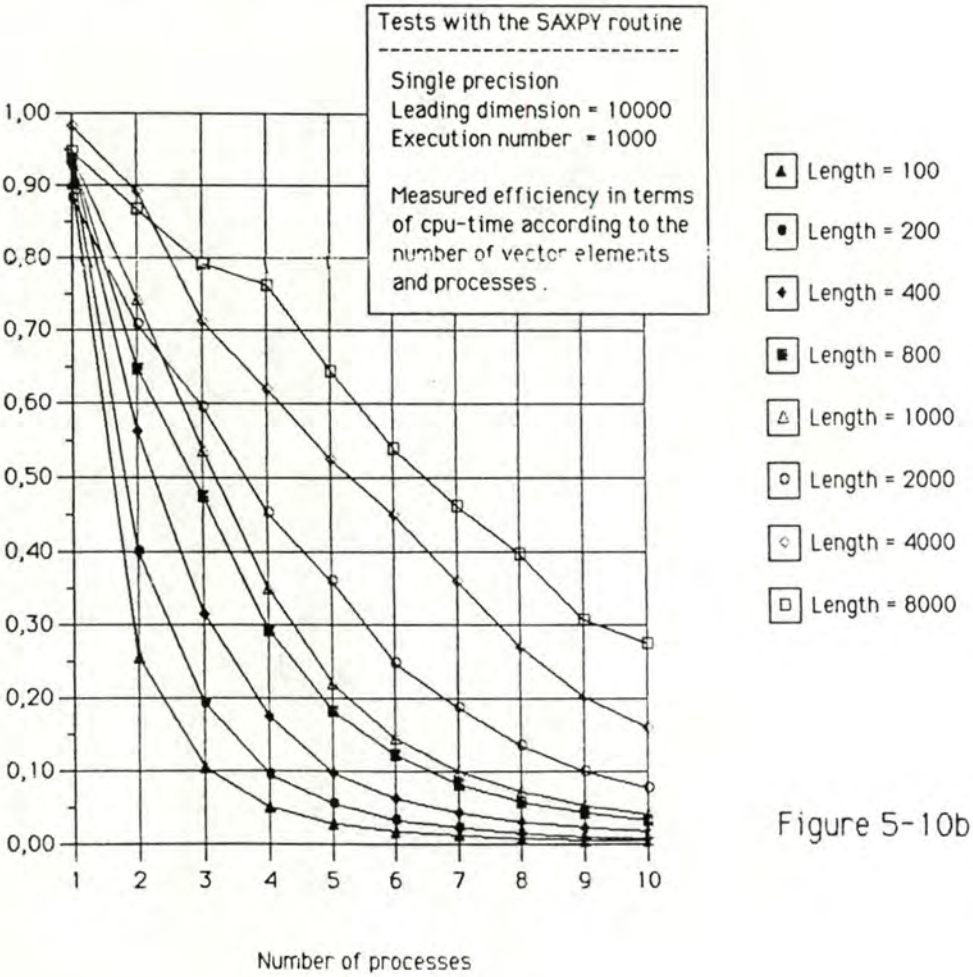


Figure 5-10b

Graphic 3

This graphic (on figure 5-11a) shows the speedup of the algorithm when it is used with many processes . The speedup is measured in terms of real-time .

The conditions and matrix sizes are the same as these explained in the graphic of figure 5-10a . The results are given in terms of real-time instead of cpu-time .

Graphic 4

This graphic (on figure 5-11b) shows the efficiency of the algorithm for the SAXPY routine . It is the complement of the previous graphic .

It can be compared with the graphic of the figure 5-10b .

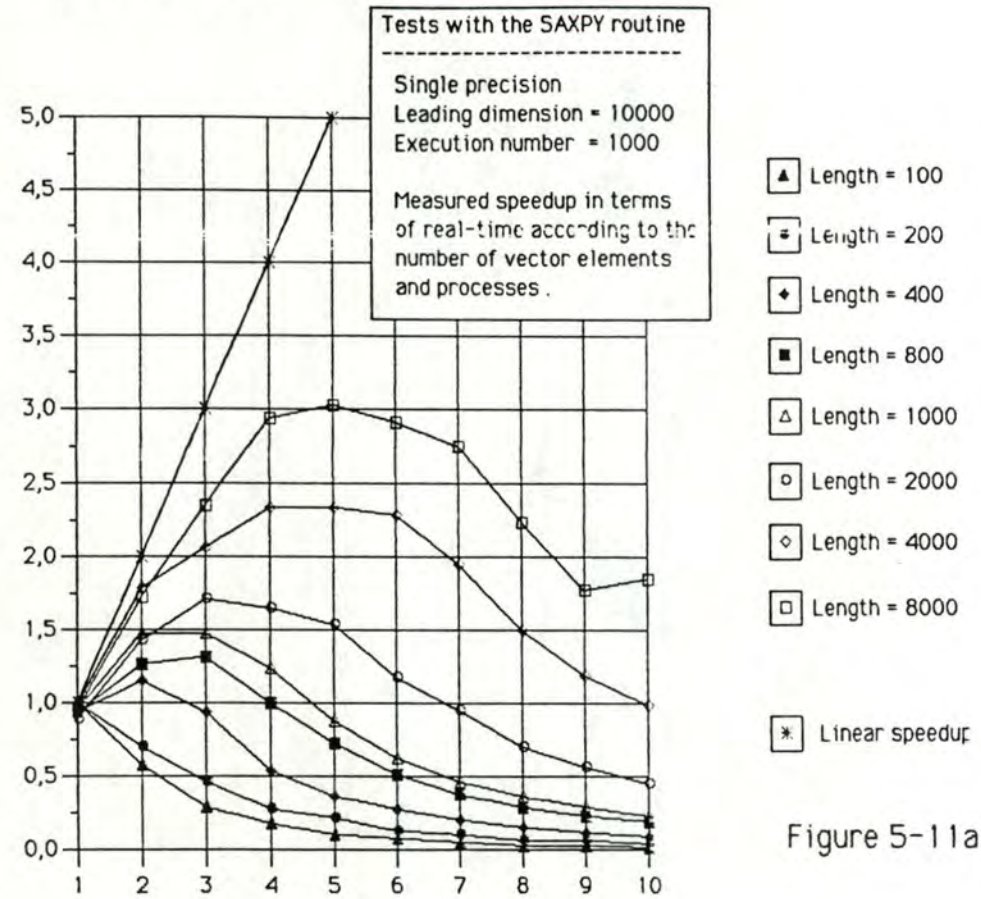


Figure 5-11a

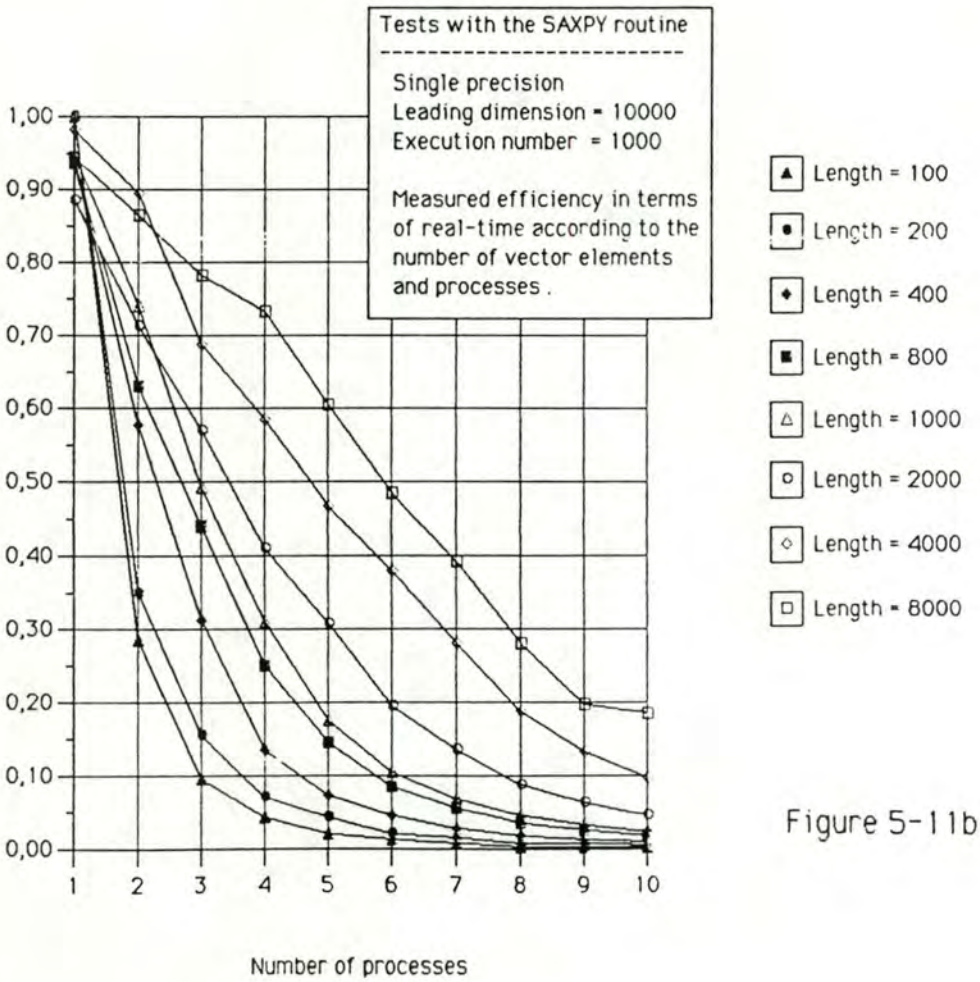


Figure 5-11b

Serie 5 of graphics

This serie contains the figures 5-12a to 5-12h provided on the next pages .

The graphics constitute an analysis of the cpu-time . We want to know more about the dilemn : How is it possible that , while the size of the parallel task decreases with the number of processes , the cpu-time is not always decreasing .

We got these results from the general tables recorded and computed while running the tests as decribed earlier .

For these graphics , we fix all parameters except the number of processes and the vector length . Each graphic is related to executions with from 1 to 10 processes , and the vector length fixed . We take into account the following vector lengths : 100 , 200 , 400 , 1000 , 2000 , 4000 and 8000 elements .

The X axe of each graphic represents the number of processes , and the Y axe is the time in seconds . In our comparisons between the results provided by the various lengths , we do not take into account the absolute value of these times , but well the frame of the curves . Each graphic contains 3 curves , the user-time the system-time and the cpu-time . The cpu-time is the sum of user and system times .

The 8 graphics show clearly where the problem of the SAXPY routine is located . This is not particulary the problem of the SAXPY routine , but well a more general problem of PARFOR . The user-time is always decreasing , except for the very little vector lengths (100 and 200) . But apart from this , we can qualify of "exected" , the behaviour of all user-time curves for the various lengths . But the most interresting in these curves , remains the behaviour of the system-time , which is always increasing .

As a direct consequence of this , the cpu-time has a minimum value , depending on the magnitude of both curves (user and system curves) .

We can see that the user-time decreases slowly for the little lengths of the vector , and decreases more quickly for the large values . To the contrary , the system-time increases very quickly for the little values of the vector length , and decreases slowly for the high values .

We can conclude that there is some features in the PARFOR environment that makes increase the system-time very quickly . These results also show the necessity of an analysis of the PARFOR overheads . That's what we do in further steps .

Anyway , concerning the SAXPY routine , we can conclude that in the current implementation of the PARFOR environment , it is no use to work with the parallelized routine if the length is less than 400 elements . And , from there , it seems "normal" that the linpack benchmark provides very bad results because the SAXPY routine , within this program , works always on very low vector lengths (from 1 to 100 elements for the standard version of the benchmark) .

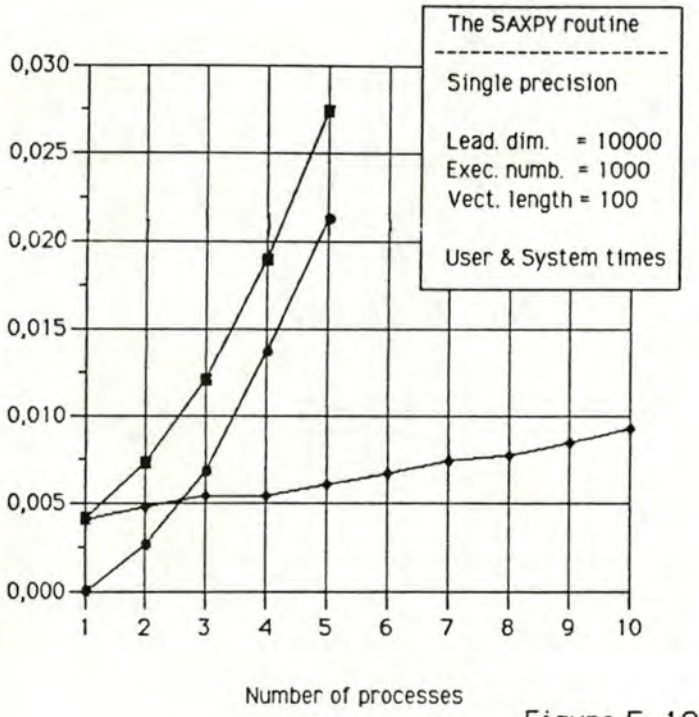


Figure 5-12a

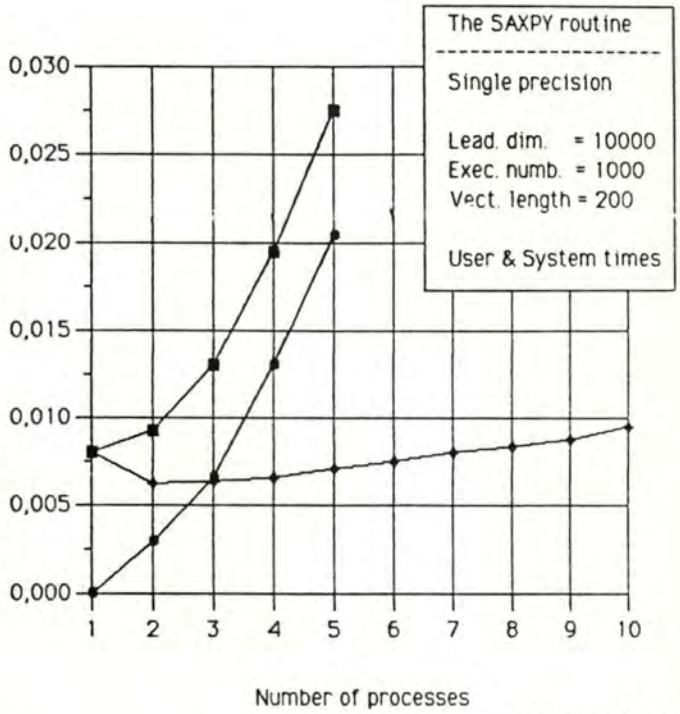


Figure 5-12b

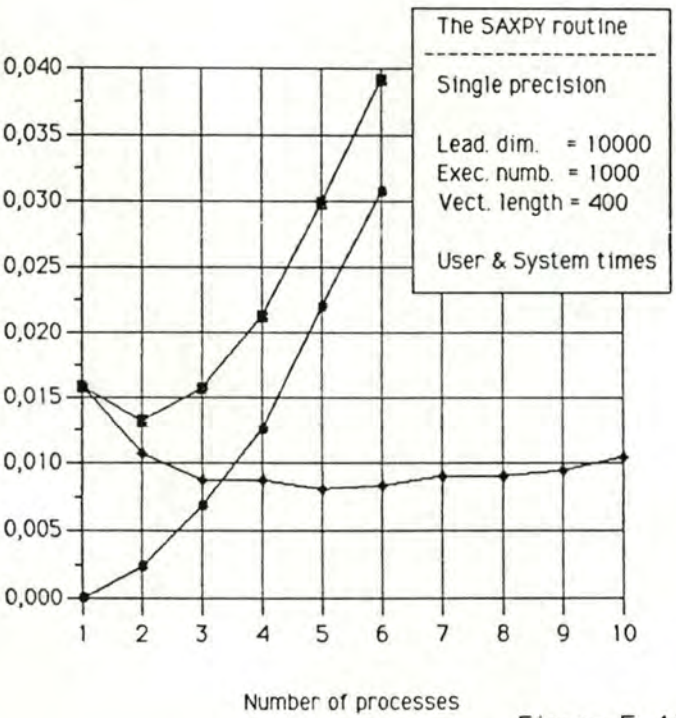


Figure 5-12c

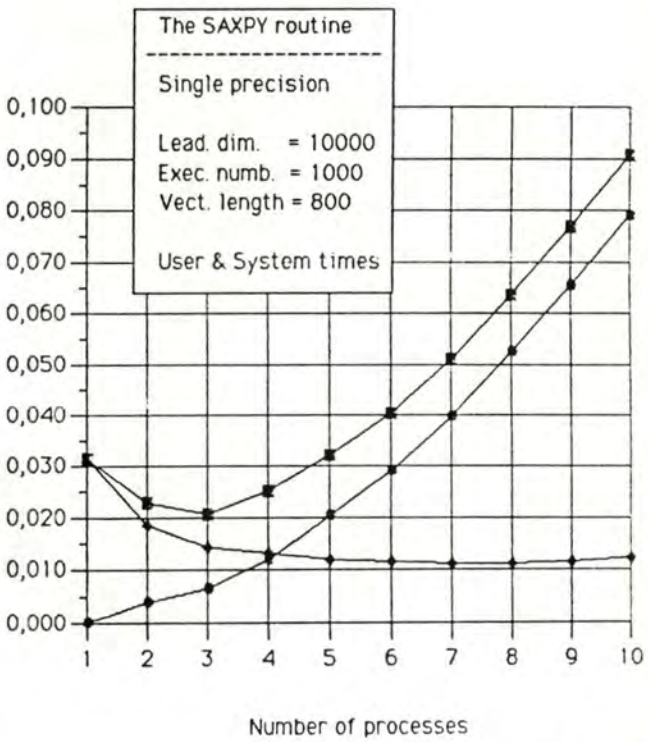
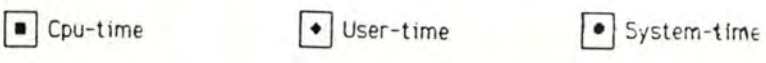


Figure 5-12d



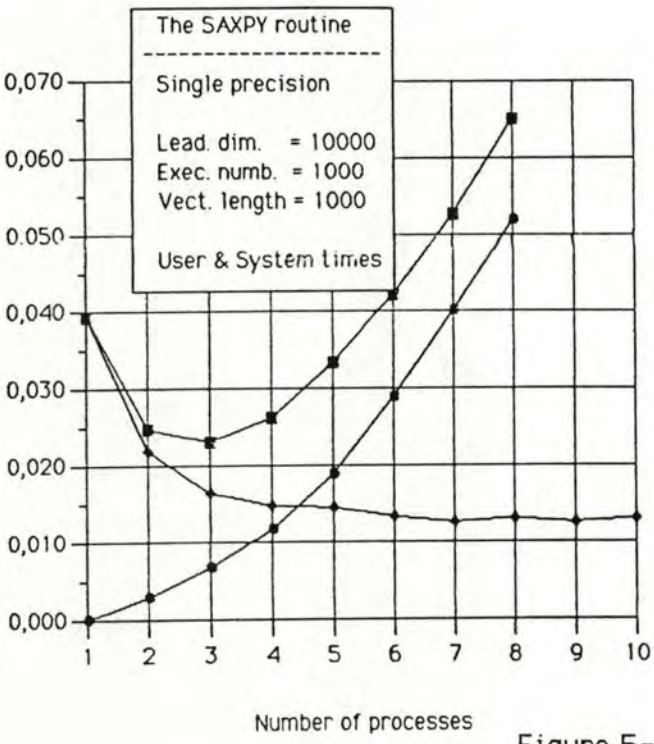


Figure 5-12e

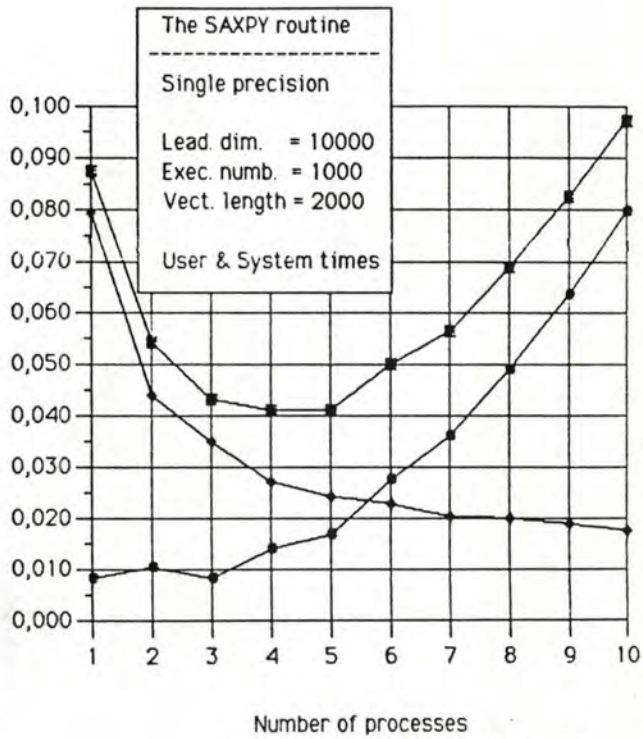


Figure 5-12f

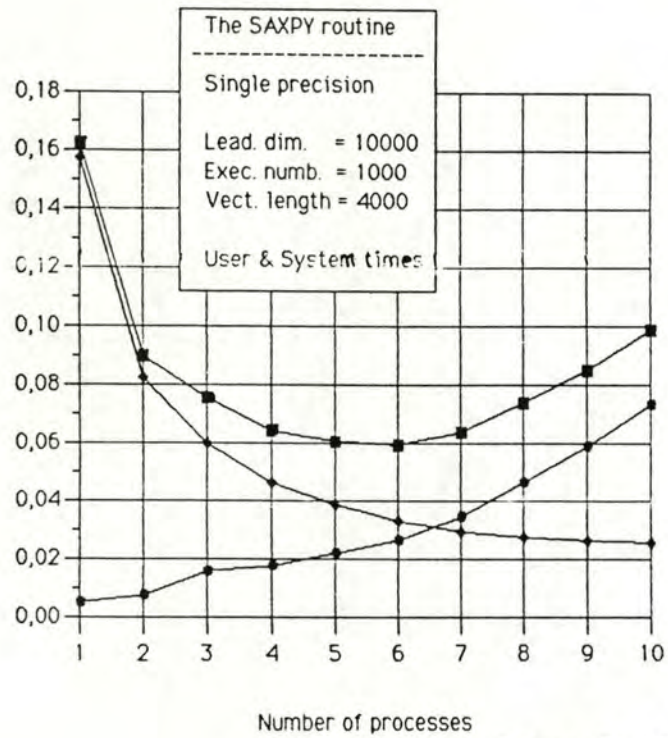


Figure 5-12g

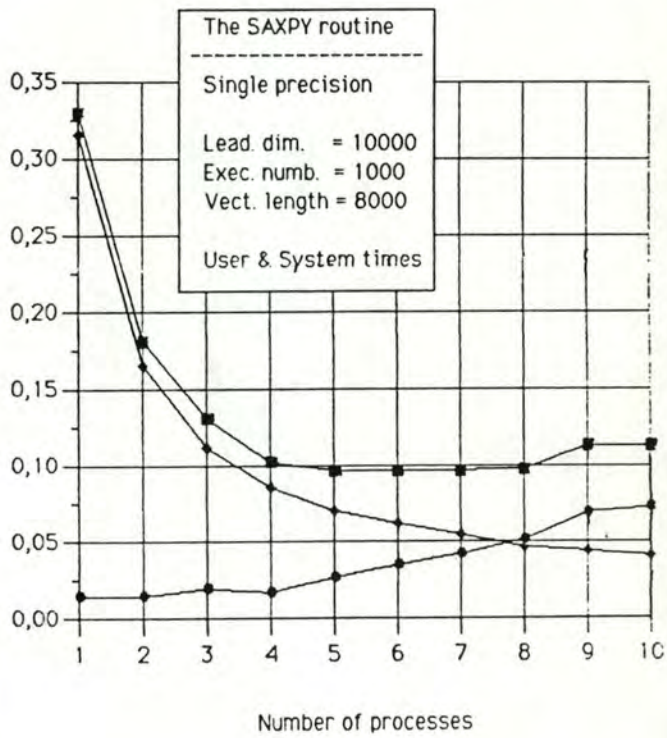


Figure 5-12h

Cpu-time User-time System-time

5.9 Tests of the PARFOR overheads

5.9.1 General description of the tests

We have explained the bad results we obtained when trying to measure the performances of the parllelized LINPACK program . In this section , we learn the behaviour of the parallel environment itself , by trying to measure the overheads due to calls to TASKIN plus the synchronization with the WAIT facility .

At the origin , we wanted to measure all kinds of overheads in a parallel program or routine . Therefore , we built a tool (described in chapter 1) to compute all results from testpoints in the program . But the major problem with this tool was coming from the timing routines . The SINIX environment , as it is also the case in all UNIX environment , does not provide any time routine with a greater resolution than 20 miliseconds . So , with such a low precision , it was not possible to measure interresting things , and we had to discard this tool .

Anyway , this problem with the time routine can not be avoided , implying that the only solution is the modification of the way the tests are made . That's why , instead of providing results for each kind of overhead (initialization and termination) , we provide a global overhead involving the global execution of a parallel program . With this view , we got interresting results less detailed , but more stable . All measurements are made in the main program .

For these tests , we provide results for the 5 versions of PARFOR , available at the present time .

5.9.2 Scheme of the tests

The measures for this kind of tests are taken in the following way :

```
Time(1)
.
Do loop i = 1 , ntimes
    .
    npar = NTASKS()
    Do loop2 k = 1 , npar
        .
        Execute the entire parallel section with the
        TASKIN facility calling the parallel work subroutine
        .
    End of Do loop2
    .
    Direct call to the parallel work subroutine
    .
    Call WAIT() facility for synchronization
    .
End of Do loop
.
Times(2)
```

The entire parallel section is a loop over a call to the TASKIN facility with the number of processes that are generated according to the -NTASKS= option at the run time . The parallel work subroutine called each time by the TASKIN , is an empty subroutine with a different fixed number of dummy parameters for each test . The parallel work subroutine performs only a call-and-return . This global time measurement ensure that we measure exactly the additional overheads present in the execution time of a parallel program .

The ntimes specifies the number of executions for calculating the results . These results are undependent from the number of iterations , because they are normalized ; but the number of iterations acts as a factor of quality on the results provided .

5.9.3 Result tables and fixed parameters

For these tests , we provide results in tables . Each table is a serie of executions of the tests with a fixed number of parameters passed to the subroutine called by the TASKIN function . Each serie of tests provides results for the execution of the test programs with from 1 to 10 processes .

We build our tests so that the number of parameters passed to the subroutine has a logarithmic variation : 0 , 1 , 2 , 4 , 8 , 16 , 32 and 64 parameters .

For these tests , we use the first tool originally described that we designed for measuring the performances of parallel programs , but we measure only in the main program .

5.9.4 Results of the tests

Tests with the version 1 of PARFOR

Figure 5-13 shows the distribution of the system and user times . Figure 5-14 shows the variation of the user-time for executions with various numbers of processes . Figures 5-15a to 5-15d show the mean repartition of the various times for various numbers of parameters passed to the parallel subroutine , according to the number of processes .

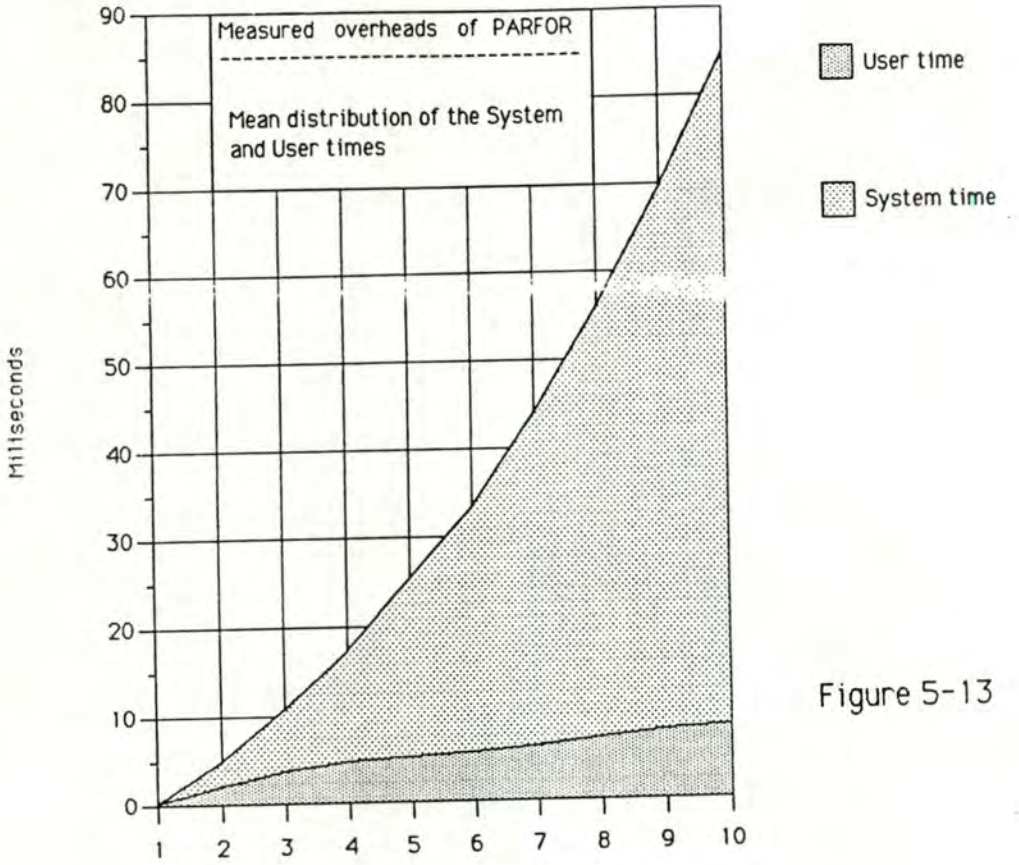


Figure 5-13

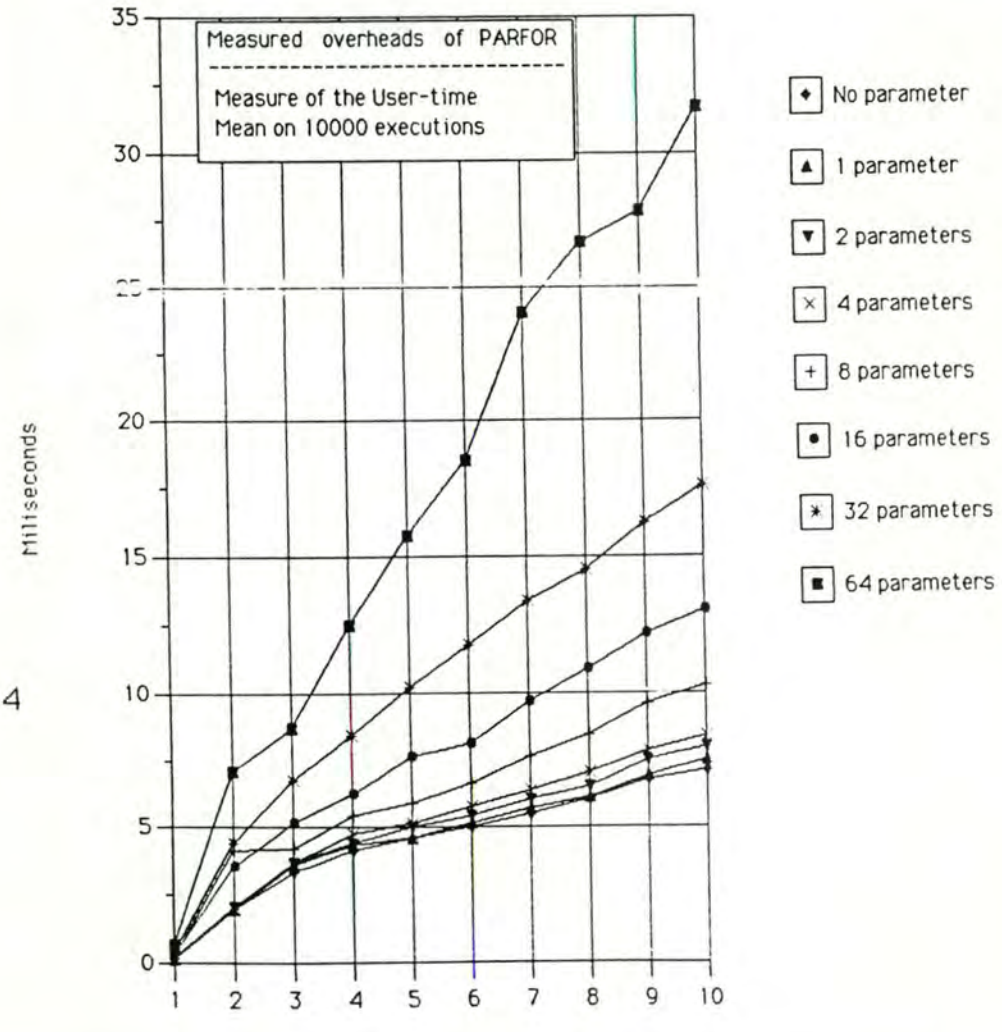


Figure 5-14

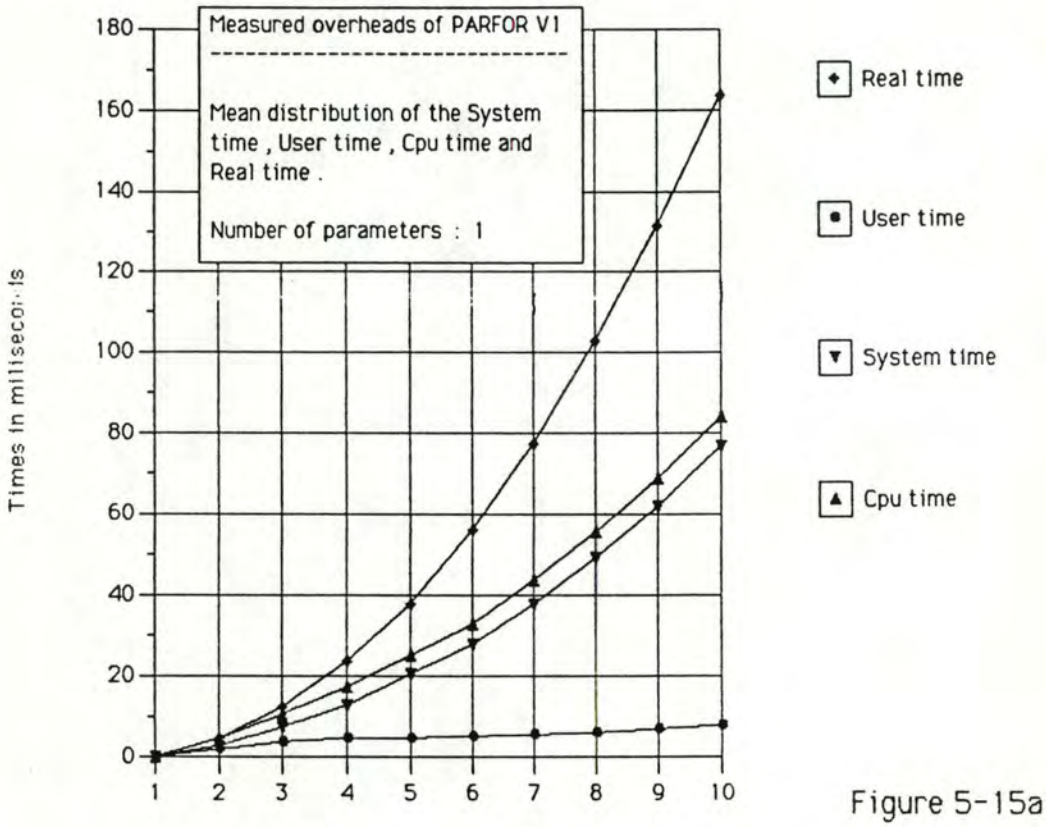


Figure 5-15a

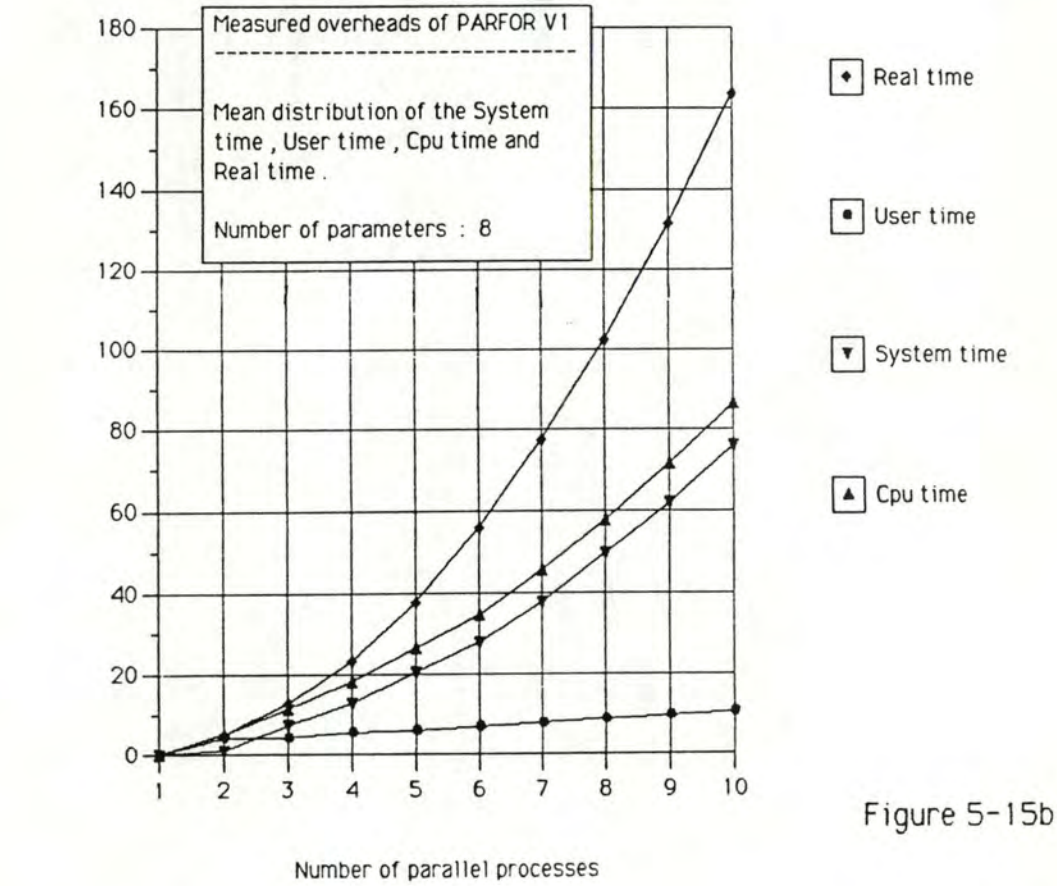


Figure 5-15b

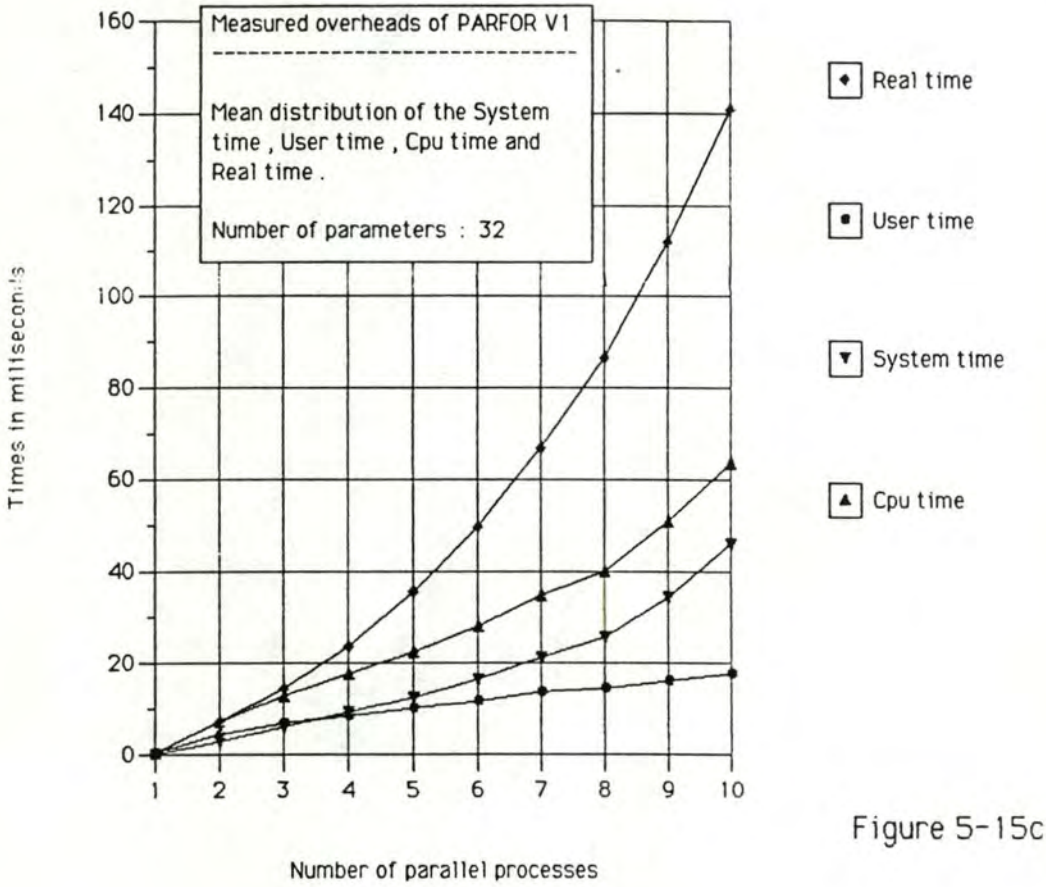


Figure 5-15c

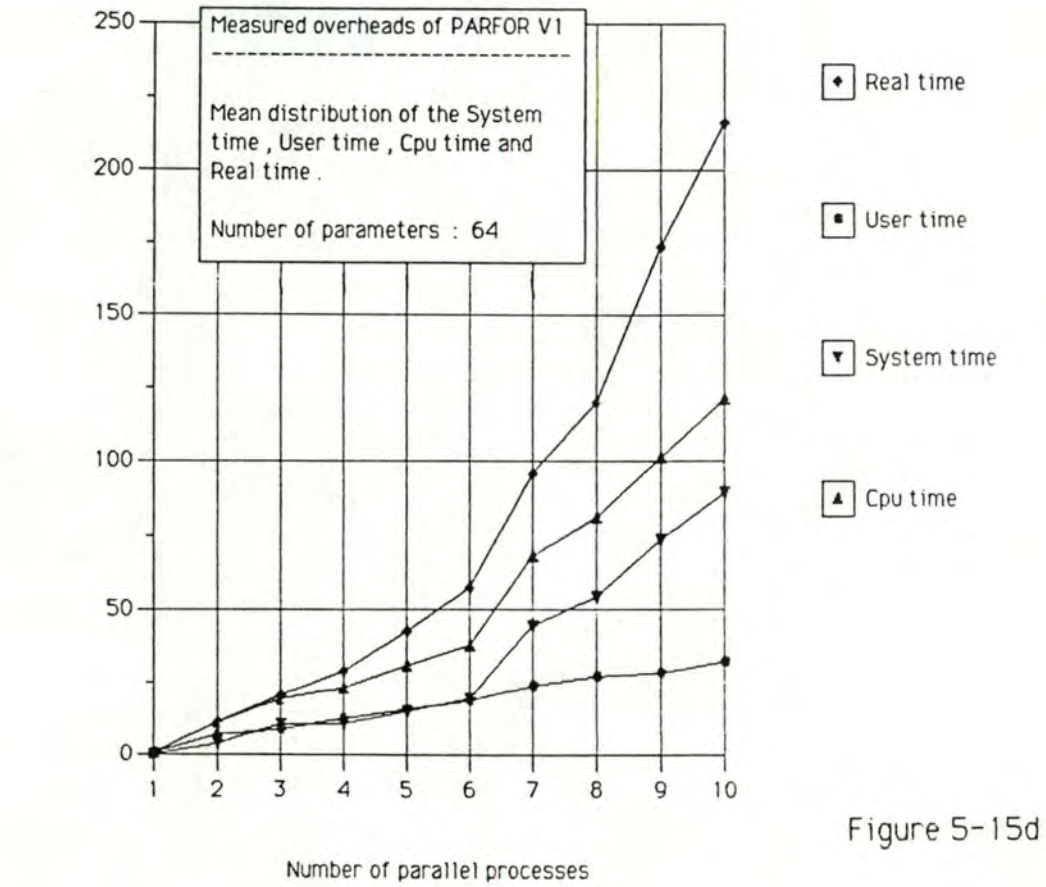


Figure 5-15d

We are surprised to observe that , when the number of parameters is varying , the system-time does not or nearly not vary according to this number of parameters passed . Instead , it is close to a constant for a fixed number of processes .

To the contrary , the user-time consumed is clearly dependent on the number of parameters passed . This is true if we know that the actual parameters are copied to the shared memory , in a loop in the C driver program .

Another interresting result is the amount of cpu-time consumed for the execution with many processes . More the number of processes increases , more the amount of cpu-time consumed is high . However , if we consider the decomposition of the cpu-time between the system and the user times , we can observe that the user-time remains very low , while the amount of system-time is increasing very quickly .

The absolute value of the cpu-time according to the various numbers of parameters remain anyway very high . This is the main reason why the performances of the parallelized program are very low . The overheads of PARFOR , in this version 1 of the implementation , are high in absolute value .

This very bad behaviour of PARFOR in terms of cpu-time lead us to reconsider the way the processes receive their part of the job from the main program . According to these results , the messages transfer facility of the IPC system V is very slow . The only system routine called are the routines concerning the messages (msgrcv and msgsnd) . In the next versions of PARFOR , we avoid these expensive calls .

According to these results , PARFOR version 1 is interresting to use to the customer point of view , depending on the size of the parallel sections that he has to create . In most of the cases , for a single precision , the mean critical size of a parallel section should perform a work of at least the following values (in miliseconds) , depending on the number of processes that must be created :

Number of parameters of the subroutine : 16
Number of executions for these results : 10000

Number of procs	Total cpu time	Equivalent Flops number	
		Single prec.	Double prec.
1	0,28	13	11
2	5,70	263	211
3	12,75	587	472
4	18,75	863	694
5	27,68	1274	1025
6	36,48	1678	1350
7	47,57	2188	1761
8	60,59	2788	2242
9	79,16	3642	2929
10	88,32	4063	3268

With this version of PARFOR , the overheads are very high . They would restrict the number of applications that can be parallelized in this environment .

The third and fourth columns provide the number of equivalent floating point operations that the machine should execute to face the overhead times provided in the column2 . These numbers are computed by the formula :

$$\text{Flop number} = \frac{\text{overhead time}}{\text{time to execute 1 flop}}$$

The time to execute 1 flop is given by the result of the sequential original LINPACK benchmark , and has the value 1ms/46 for single precision and 1ms/37 for double precision .

If the number of operations to perform in a parallel process is lower than this number , the sequential environment is more usefull in terms of speedup . These numbers are theoretical values , according to the measures

of the overheads and the estimated power of the cpu . In the reality , there are some little variations around these values . For example , the values that we measured within the tests of the SAXPY routine , are following this rule until 4-5 processes , and then , the overheads in this routine are greater than the theoretical values that we provide here . But this is quite understandable if we consider , in these theoretical values , that we have the same number of processors than the number of processes . We must remember that our machine has only 6 cpus , implying that we should consider the values until 6 processes only .

Tests with the version 2 of PARFOR

This second version of PARFOR , according to our tests , provides better results in terms of both cpu-time and real-time .

We observe that , when the number of parameters is varying , the system-time is quasi constant . To he contrary with the previous version of PARFOR , (version 1) , the system-time is now a very low part of the total cpu-time , quasi negligible . It increases with the number of processes , but of a very small amount , and only when the number of processes is greater than the available number of processors .

The user-time consumed , as it was the case for version 1 , remains dependent on the number of parameters passed . The explanation we had for version 1 is still valuable for this second version .

The amount of cpu-time consumed for executions with many processes increases , but in a reasonable way , i.e. in proportions with the number of parameters passed to the parallel subroutine .

The absolute value of cpu-time according to the various numbers of parameters are now reduced . This should lead to better performances of the parallelized programs with this second environment .

This better behaviour of PARFOR is mainly due to the shared memory and a busy loop that replace the message system calls .

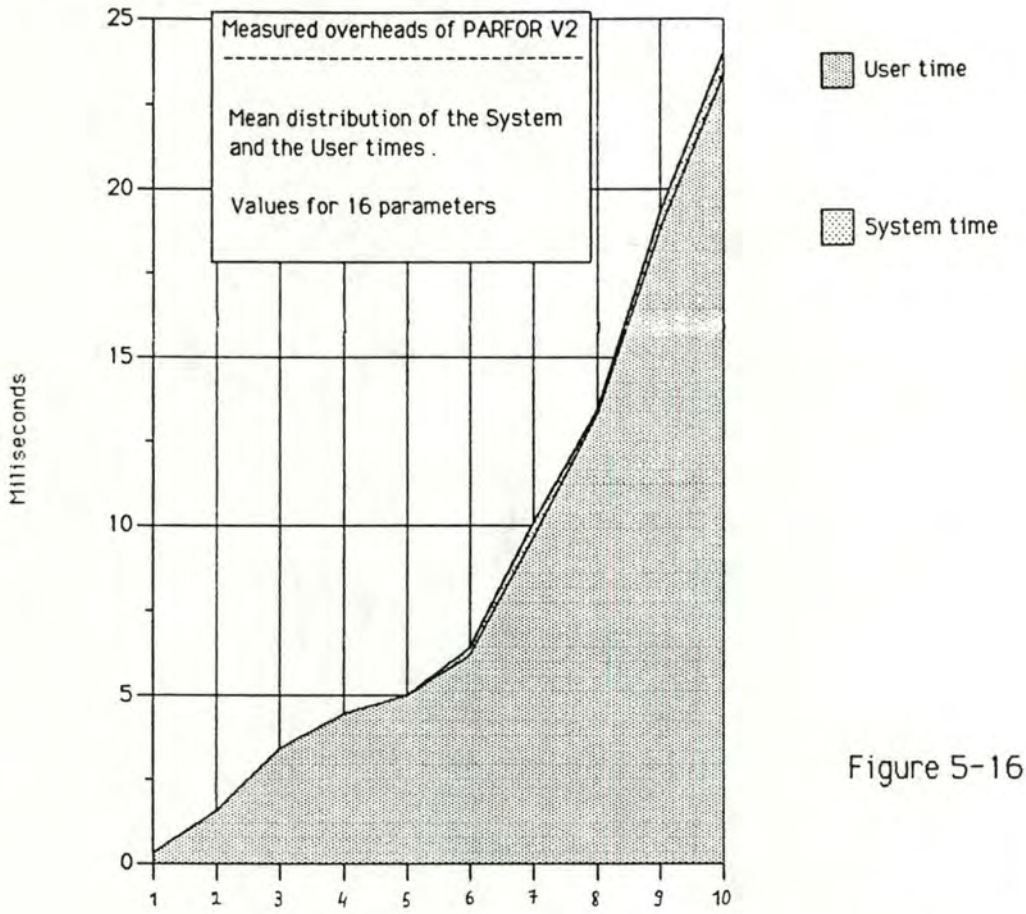


Figure 5-16

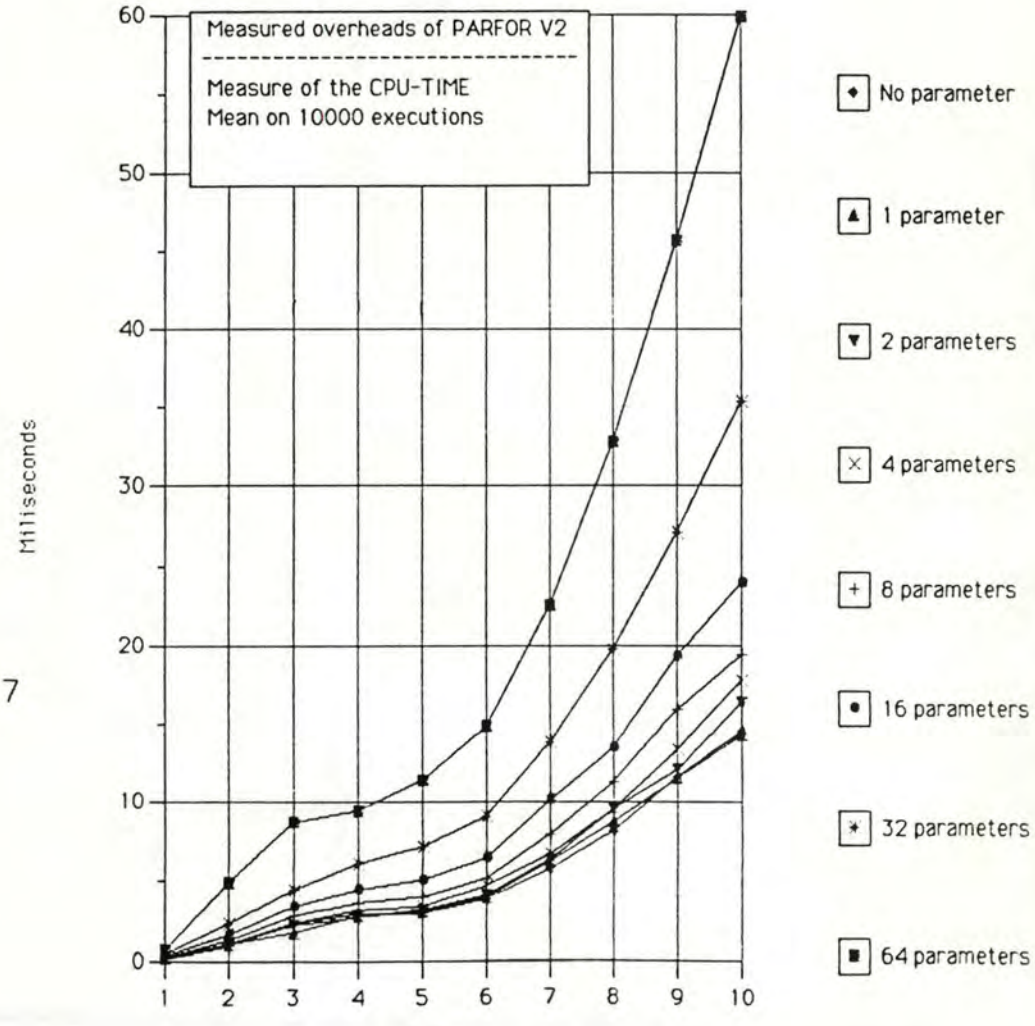


Figure 5-17

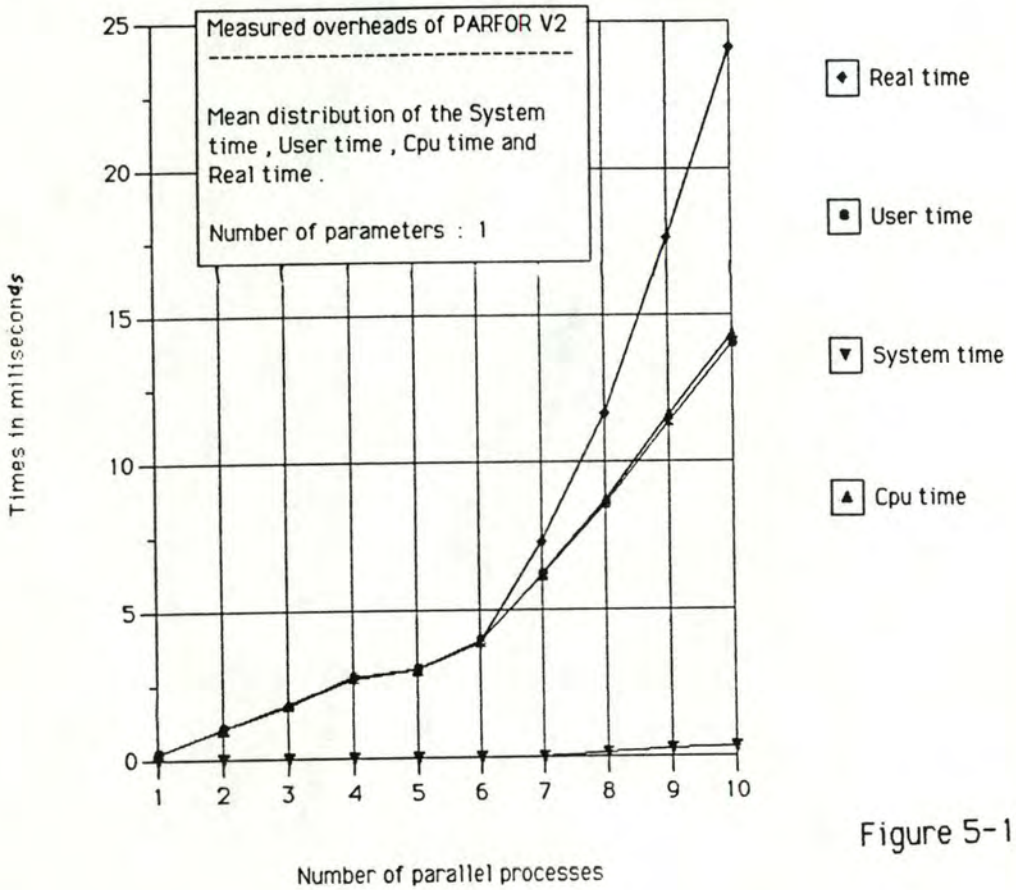


Figure 5-18a

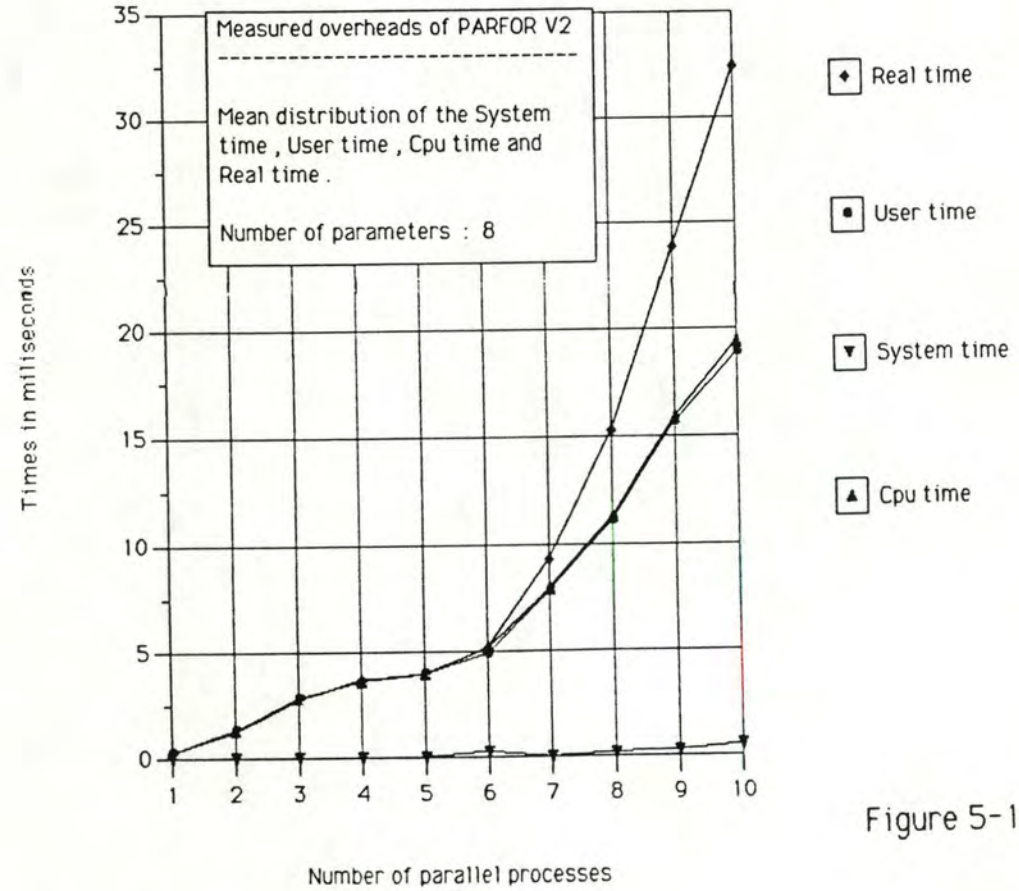


Figure 5-18b

Figure 5-16 shows the repartition of the cpu-time between the user and system times . Notice the very little part of the system time .

Figure 5-17 shows the variation of the user-time for executions with various numbers of processes .

Figures 5-18a to 5-18b show a mean distribution of the various times for the tests . This is for the tests with 8 parameters passed to the parallel subroutine .

For the first version , we provide some summary tables showing approximated equivalent costs of the overheads introduced by the use of the PARFOR . We can reproduce the same tables calculated in the same way , for this second version . We have :

Number of parameters of the subroutine : 8
Number of executions for these results : 10000

Number of procs	Total cpu time	Equivalent Flops number	
		Single prec.	Double prec.
1	0,23	11	9
2	1,28	59	47
3	2,83	130	105
4	3,64	167	135
5	3,93	181	145
6	5,16	237	191
7	7,92	364	293
8	11,31	520	418
9	15,89	731	588
10	19,42	893	719

It is true by these results , that this version allows a middle granularity of the program to be taken into account by PARFOR . The costs of the TASKIN calls are in another order of magnitude , lower for this version . The busy loop that replaces the calls to the message system in the implementation , is less expensive .

Note however that the behaviour of the processes is different for versions 1 and 2 . In the first version , when a process is waiting , it does not consume any cpu-time . In the second version , it remains in a busy loop . But the results presented here lead to think that , even if there is a busy loop , the results claim in favour of this second version .

Tests with the version 3 of PARFOR

This third version of PARFOR is an attempt to substitute all hardware atomic lock memories of the implementation , by software system V semaphores , based mainly on the Dekker's algorithm .

The results for this version are provided only for the execution of the test with no parameter . We did not go further in these tests , because it appeared immediately that this solution was the worst possible .

Figure 5-19 shows the distribution of the system and user times . Figure 5-20 show the mean repartition of the various times for no parameter passed to the parallel subroutine , according to the number of processes .

The system V semaphores is very slow and then , it is also too expansive to execute all tests with this solution .

The first graphic shows the repartition of the cpu-time into the user-time and the system-time . All the system-time is consumed by the management of the software semaphores . The user-time remains unsignificant in the total cpu-time .

The second graphic shows the 4 curves , real-time , cpu-time user-time and the system-time . It is clear on the graphic , that this solution is to be rejected .

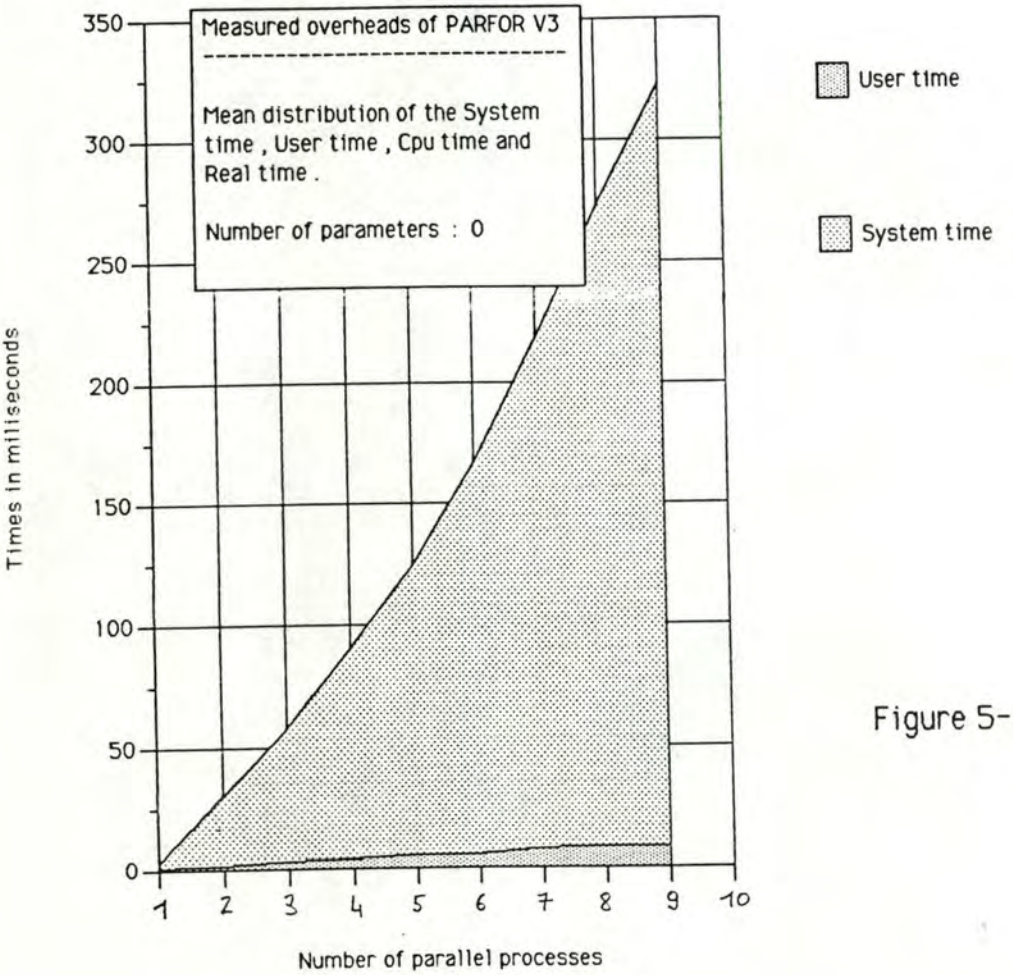


Figure 5-19

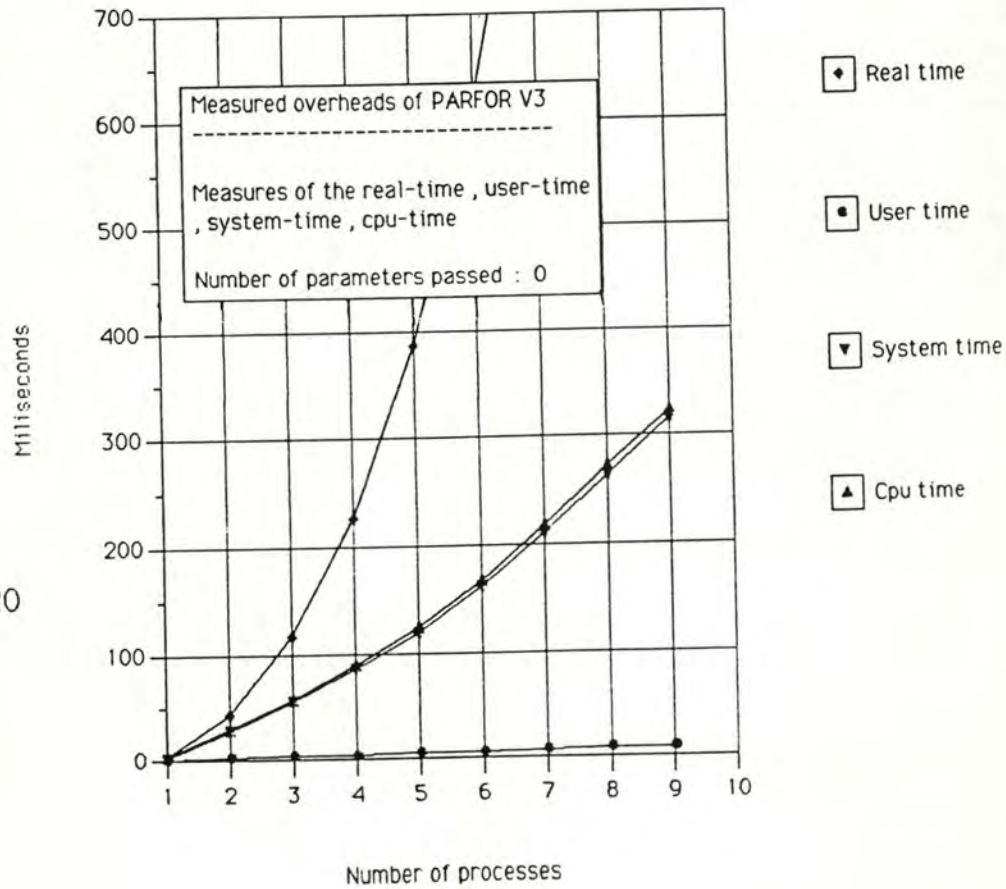


Figure 5-20

Tests with the version 4 of PARFOR

This version of PARFOR shows better results than the version 2 . However , exactly the same remarks from the version 2 can be transported for this version .

Figures 5-21a to 5-21b show the mean repartition of the various times for various numbers of parameters passed to the parallel subroutine , according to the number of processes .

Remember that here , we modified the environment such a way that the WAIT facility is implemented without calls to the parallel library previously necessary to access the atomic lock memories .

We also compute the approximated equivalent number of floating point operations for the time results of this solution :

Number of parameters of the subroutine : 8
Number of executions for these results : 10000

Number of procs	Total cpu time	Equivalent Flops number	
		Single prec.	Double prec.
1	0,20	10	8
2	0,90	42	33
3	1,60	74	59
4	2,20	101	81
5	3,20	147	118
6	4,50	207	167
7	8,30	381	307
8	13,30	611	492
9	19,10	879	707
10	29,10	1339	1077

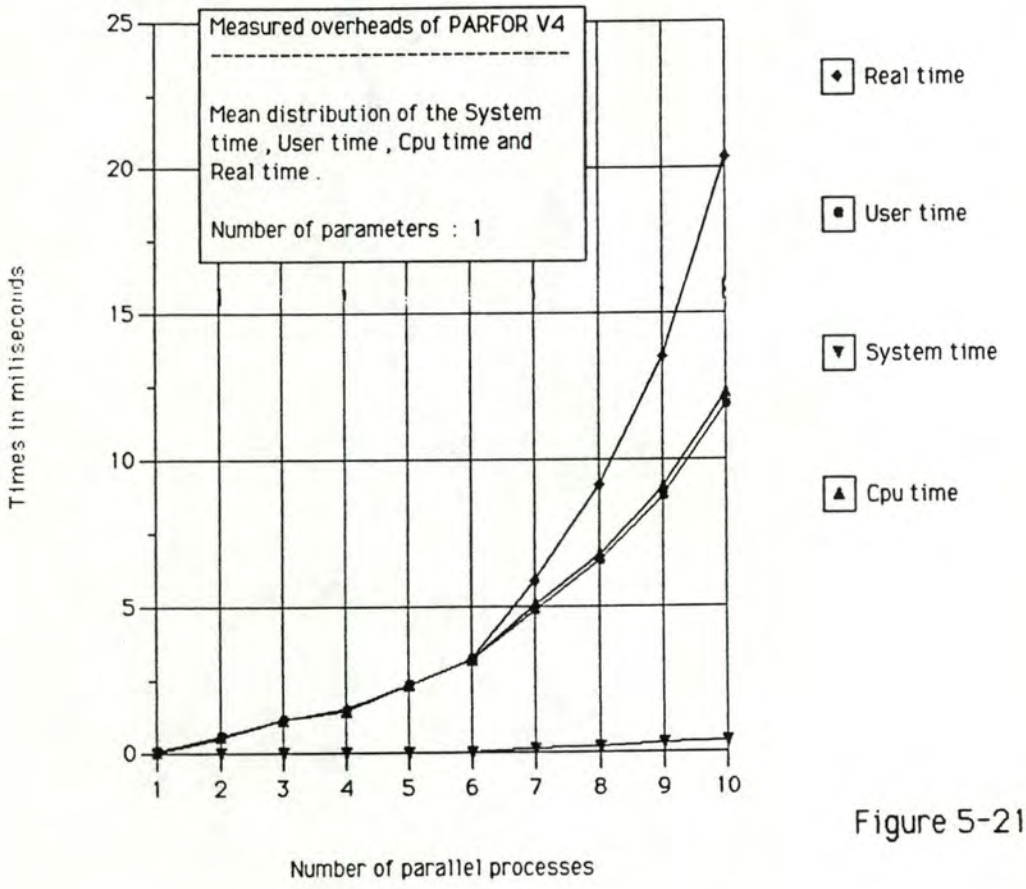


Figure 5-21a

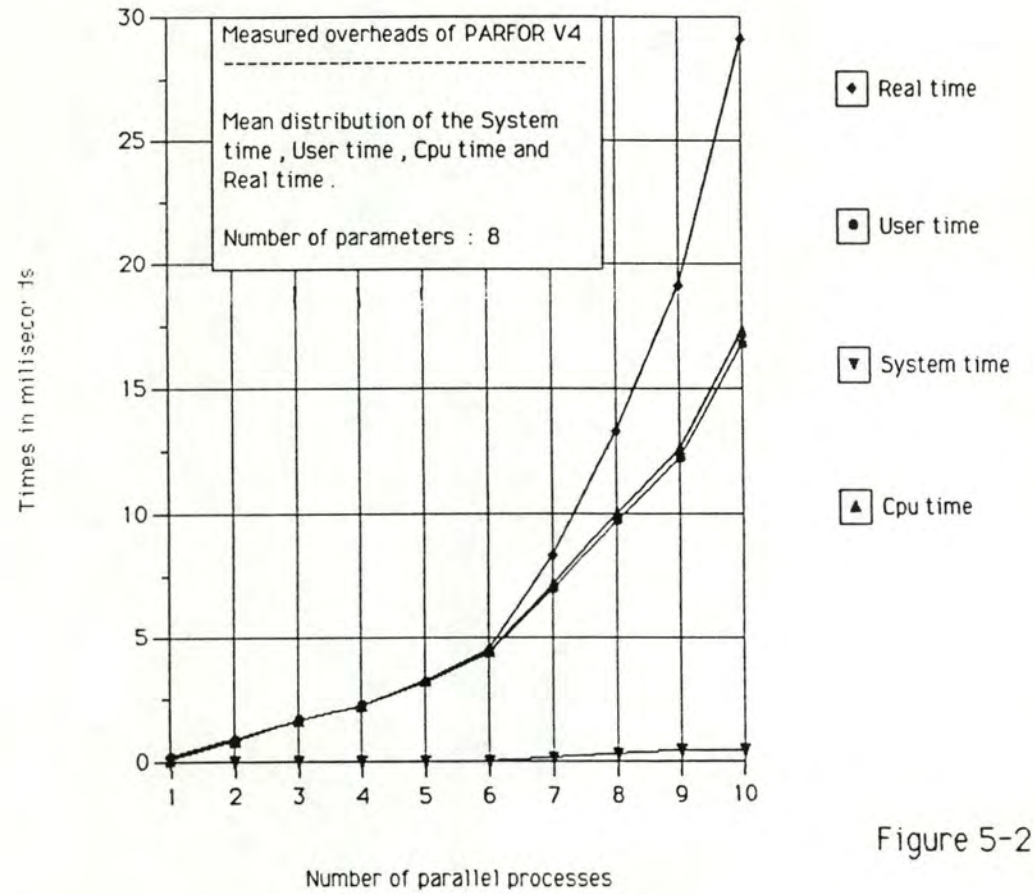


Figure 5-21b

These equivalent floating point numbers are better than those of the version 2 of PARFOR . As we can see when comparing these tables , this is always true when the number of processes is lower than the available number of processors (6 processors on our test machine) .

The conclusion for this version is good anyway .

Tests with the version 5 of PARFOR

This is the last version of PARFOR that we implemented until now . The target of these multiple versions is always the reduction of the costs when calling the TASKIN and WAIT facilities . As we have explained in the chapter describing these various versions , this one is implemented without any lock .

Figures 5-22a & 5-22b show the mean repartition of the various times for various numbers of parameters passed to the parallel subroutine , according to the number of processes .

The results of these tests show similar results to those provided for the previous version . The comments made for the version 2 are also still valuable for this version . The frames of the curves that we draw are close to those of the previous versions .

Hewever , the suppression of the remaining calls to the atomic lock facilities benefit to the performances of the TASKIN and WAIT facilities .

Always in the same way , we compute the equivalent floating point operation number for the various number of processes . This gives us the following table :

Number of parameters of the subroutine	: 8
Number of executions for these results	: 10000

Number of procs	Total cpu time	Equivalent Flops number	
		Single prec.	Double prec.
1	0,10	5	4
2	0,70	32	26
3	1,10	56	41
4	1,40	64	52
5	1,90	87	70
6	2,40	110	89
7	4,90	225	181
8	9,80	451	363
9	16,40	754	607
10	31,40	1444	1162

With such values , the PARFOR environment becomes quite interresting to use for low granularity parallel programs . But the costs are still too high for some applications . Linpack is such a kind of application which has a rather fine granularity , that can only be exploited with an environment having a very low total overhead . Even with these results of the version 5 of PARFOR , the results of linpack would not be very good .

Until now , this version is the best implementation . We can also compare the overheads in terms of floating point operations , with the overheads of the original version . The reduction of the costs is drastic .

The graphics show that a similar frame of the curves to the version 4 and version 2 . Mostly , the absolute values are decreased in this version 5 .

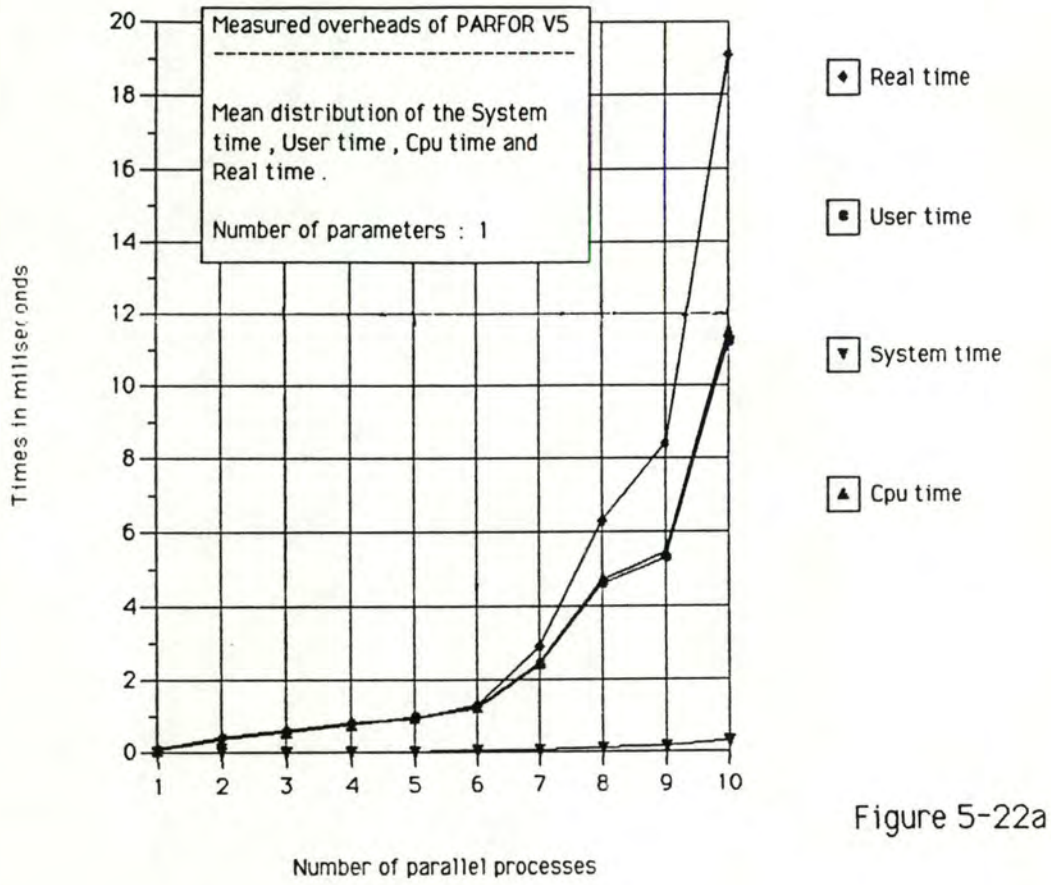


Figure 5-22a

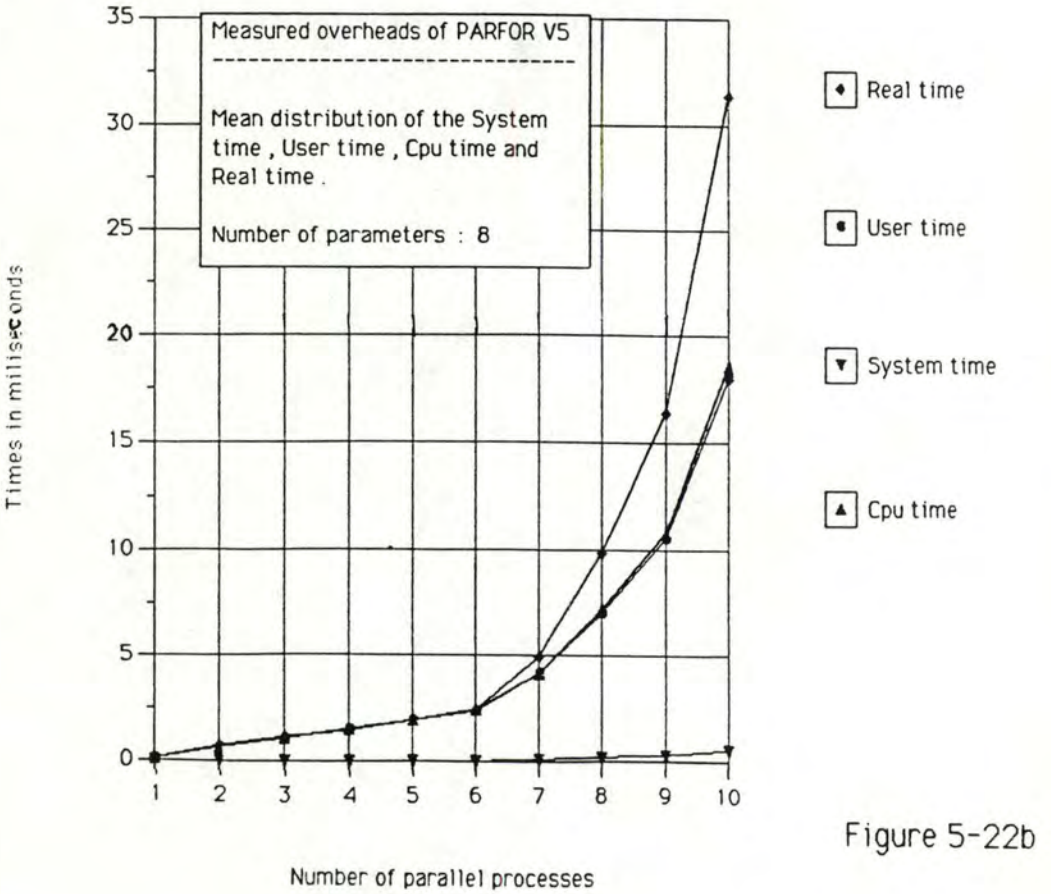


Figure 5-22b

5.10 Tests with the SGEFA routine

5.10.1 General description of the tests

In this section , we describe the tests made with the SGEFA routine . The most important thing is not that the results are provided by SGEFA , but well that the SGEFA routine is a good example for demonstrating what is practically possible with PARFOR . A similar test is done for FORCE .

The test program with SGEFA was written in 3 versions . The first version is parallelized at the innermost level with the standard tools of PARFOR . The second version is parallelized at a higher level . And the third version is parallelized at the highest possible level in the PARFOR environment .

Most of these tests have been made in 5 versions of PARFOR described earlier .

Eventually , a FORCE version of the parallelized program at the highest level was written and testsed , and the time results are compared with those of the PARFOR environment .

5.10.2 Scheme of the SGEFA routine

The SGEFA routine , in its sequential version , is structurated in the following way :

Do loop over the pivots

.

Search the absolute maximum value in the column to find the new pivot .

.

Swap the pivot row into position

.

Take the reciprocal of the pivot

.

Reduce the non-pivot rows

.

End of loop over the pivots

5.10.3 Various parallelized versions within PARFOR

Particularities of the first parallelization

The first parallelized version of the PARFOR SGEFA program is done in its innermost routines . This parallelization was done in a very similar way as in the tests with LINPACK .

So , the calls to the subroutines in SGEFA remain unchanged , while the subroutines themselves are parallelized . In this way , the standard interface of the subroutines is preserved , except that the subroutines contain some common statements necessary for the parallel work subroutines . The calls to the TASKIN PARFOR facility are made in the subroutines that are parallelized , implying that a new parallel subroutine must also be described (the parallel subroutine that is called by the TASKIN facility) .

Scheme of the first parallel algorithm

```
Do loop over the pivots
    .
    Call izaMAX to search the absolute maximum value
        .
        Call TASKIN with the parallel subroutine
        and the parameters for each process
        .
    End izaMAX
    .
    Determine pivot and its inverse
    .
    Call SSCAL to compute the multipliers
        .
        Call TASKIN with the parallel subroutine
        and the parameters for each process
        .
    End SSCAL
    .
    Row elimination with column indexing
        .
        Call SAXPY to modify the rows
            .
            Call TASKIN with the parallel subroutine
            and the parameters for each process
            .
        End SAXPY
        .
    End row elimination
    .
End loop over the pivots
```

Particularities of the second parallelization

In the second version , the parallelization is made at a higher level . The standard interface is lost . The subroutines are directly parallelized in the body of SGEFA , instead of inside the subroutines themselves . That is why the standard interface of the subroutines called is lost . These subroutines are replaced by the calculations of the boundaries for the parallel work ,

and a loop to call the TASKIN facility , the number of times that there are parallel processes . Note that in this solution , the number of times that the TASKIN facility is called , is not reduced . Only the number of calls to subroutines is reduced . Each original call to a subroutine is replaced by a loop over calculations of the boundaries , and over calls to the TASKIN facility .

Scheme of the second parallel algorithm

```
Do loop over the pivots
    .
    Call TASKIN with the parallel subroutine and the
    parameters for each process to search the
    absolute maximum value
    .
    Determine pivot and its inverse
    .
    Call TASKIN with the parallel subroutine and the
    parameters for each process to compute the
    multipliers
    .
    Row elimination with column indexing
        .
        Call TASKIN with the parallel subroutine
        and the parameters for each process to
        modify the rows
        .
    End row elimination
    .
End loop over pivots
```

Particularities of the third parallelization

The thirs parallelization of this algorithm is made at the higest level , implying that a minimal number of calls to the TASKIN facility is made . The body of the SGEFA algorithm is still more reduced , and contains only some initializations , and a loop to call the TASKIN facility the same

number of times that there are parallel processes . No parameters are given to the parallel called subroutine , but instead , the shared memory is extensively used . The various calculations related to the distribution of the work among the processes are made in the parallel routine with shared or private variables . The shared variables are known by all processes and must be updated only in the sequential section of a barrier , but can be readen at any time by all of them . The private variables are used for defining the boundaries , for indexes , and so on . This kind of parallelization uses FORTRAN barriers for the synchronization between the processes . As we describe in chapters 2 and 3 , the implementation of the barriers in the PARFOR environment do not require any special hardware locks , because we always consider that one of the processes is the driver and the only one allowed to execute the sequential code of the barrier .

This kind of parallelization , as we have seen in chapter 4 , is the standard way to program in the FORCE environment , while it is not the original philosophy of PARFOR .

The third parallel algorithm is a derivation adapted for PARFOR , of the original parallel algorithm drawn by professor H. JORDAN for its own FORCE environment . The main difference resides in the critical sections not available in PARFOR . However, we are sure that one of the parallel processes keeps always the control of the others when executing the barrier code .

From there , we can convert the critical sections into sequential code that can be executed in the sequential protected code of the barrier . In fact , the only differences between the FORCE version and the PARFOR version are the following :

- The FORCE version uses one critcal section . It is converted to a loop controlled by the main process of a FORTRAN barrier in the PARFOR environment .

- The FORCE version has an implicit implementation of the barriers with atomic locks , while the PARFOR version uses only FORTRAN statements .

- The FORCE version does not need to compute explicitely the boundaries of the work for each process . It is calculated automatically by the preprocessor . In the PARFOR environment , these boundaries must be explicitely calculated , and determined as a function of the process identifier .

Scheme of the third parallel algorithm (PARFOR version)

The parallel main program is built as follow :

```
Initialize synchronization barriers
.
Do for each parallel process
    .
    Call TASKIN with the parallel routine and with
    only as parameter , the identifier of the process
    .
    Direct call to the parallel routine
    .
    Call WAIT facility from PARFOR
    .
End do loop
```

The parallel subroutine is built in the following way :

```
do loop over the pivots
    .
    Search part of the pivot column for private maximum
    .
    Barrier code
        .
        Update the global maximum and record the pivot
        .
    End barrier code
    .
```

```
        Swap part of pivot row into position
        .
        Barrier code
            .
            Take the reciprocal of the pivot
            .
        End of barrier code
        .
        Reduce part of non pivot rows
        .
        Barrier code
            .
            reset global maximum
            .
        End of barrier code
        .
    End loop over pivots
```

Scheme of the third parallel algorithm (FORCE version)

The parallel algorithm is built in the following way :

```
    Do loop over the pivots
        .
        Search part of pivot column for private maximum
        .
        Critical section
            .
            Update global maximum
            .
        End critical section
        .
        Barrier code
            .
            Record pivot when all processes have finished
            .
        End barrier code
```

```
.
Swap part of pivot row into position
.
Barrier code
    .
    Take reciprocal of pivot
    .
End of barrier code
.
Reduce part of non-pivot
.
Barrier code
    .
    Reset global maximum
    .
End barrier code
.
End loop over pivots
```

5.10.4 Description of the result tables

The result tables provided contain the same informations as it is the case for the other tests . But for the present tests , we had too many tests to do , so we decided to suppress the line in the results , that provided the comparison with the parallel version of the algorithm executed as a sequential program . This execution was interresting for measuring the overheads introduced by the parallel version of the algorithm , compared with the original sequential execution .

5.10.5 Scheme of the results

The measures for this kind of tests are taken in the following way :

```
        Initializations
        .
        Time(1,1)
        .
        do loop : i = 1 , ntimes
              .
              execute the sequential algorithm
              .
loop:   continue
        .
        Time(1,2)
        .
        Initializations
        .
        Time(2,1)
        .
        Do loop2 : i = 1 , ntimes
              .
              execute the parallelized algorithm
              .
loop2:  continue
        .
        Time(2,2)
        .
        Computations and prints
```

5.10.6 Fixed parameters for the executions

The fixed parameters for the results are the following :

Parameter 1 : Leading dimension of the matrix

This parameter corresponds to the static enclosing matrix size fixed at the compile time . This dimension corresponds to the maximum size of a linear system that can be solved by the algorithm . This parameter has the same meaning as the parameter of the same name that has been used in the LINPACK tests

Parameter 2 : Matrix dimension

This parameter is the size of the linear system to be solved . This number must be lower than the leading dimension .

Parameter 3 : Sequential execution number

This parameter is the number of executions of the sequential algorithm on which the mean results provided in the tables are computed .

Parameter 4 : Parallel execution number

This parameter is the number of executions of the parallel algorithm on which the mean results provided in the tables are computed .

These parameters are valid for all results of the tables . We made vary them in the following ranges :

Leading dimension	: 501
Matrix dimension	: 25 . . 500
Sequ. exec. number	: 3 or 1 , depending on the matrix size
Parall. exec. number	: 3 or 1 , depending on the matrix size
Processes	: 1 . . 6

From the tables , we draw some graphics , showing the critical results . All these results are available in the appendix 4 .

5.10.7 Results of the tests with the first level of parallelization

The first version of the SGEFA algorithm was not used for the tests . The main reason of this attitude is that this version was already extensively used within the tests of the LINPACK benchmark . So , we know that the results provided by the parallelization of this algorithm by this level , are very bad . For more informations , refer to the section describing the tests we made first with LINPACK .

5.10.8 Results of the tests with the second level of parallelization

Tests with the version 1 of PARFOR

The graphics of figures 5-23 a & b provided show the bad performances of PARFOR when the TASKIN facility is invoked very often for very small parallel jobs to perform .

The SGEFA calls very often the TASKIN facility for executions of very little jobs , so , these are the circumstances in which PARFOR has the badest behaviour . This is confirmed by the curves showing that the speedup is better when the size of the matrix (and thus the parallel task) is greater .

The graphics are nearly the same for the Real-time and the Cpu-time .

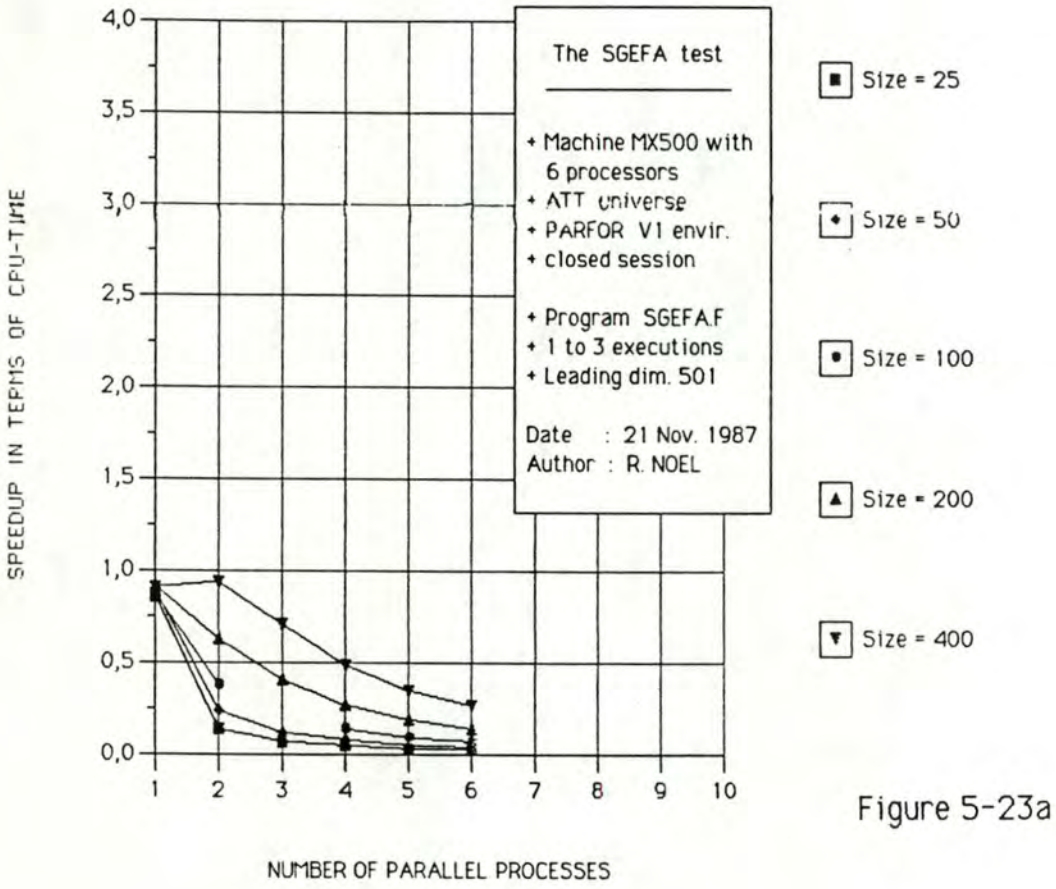


Figure 5-23a

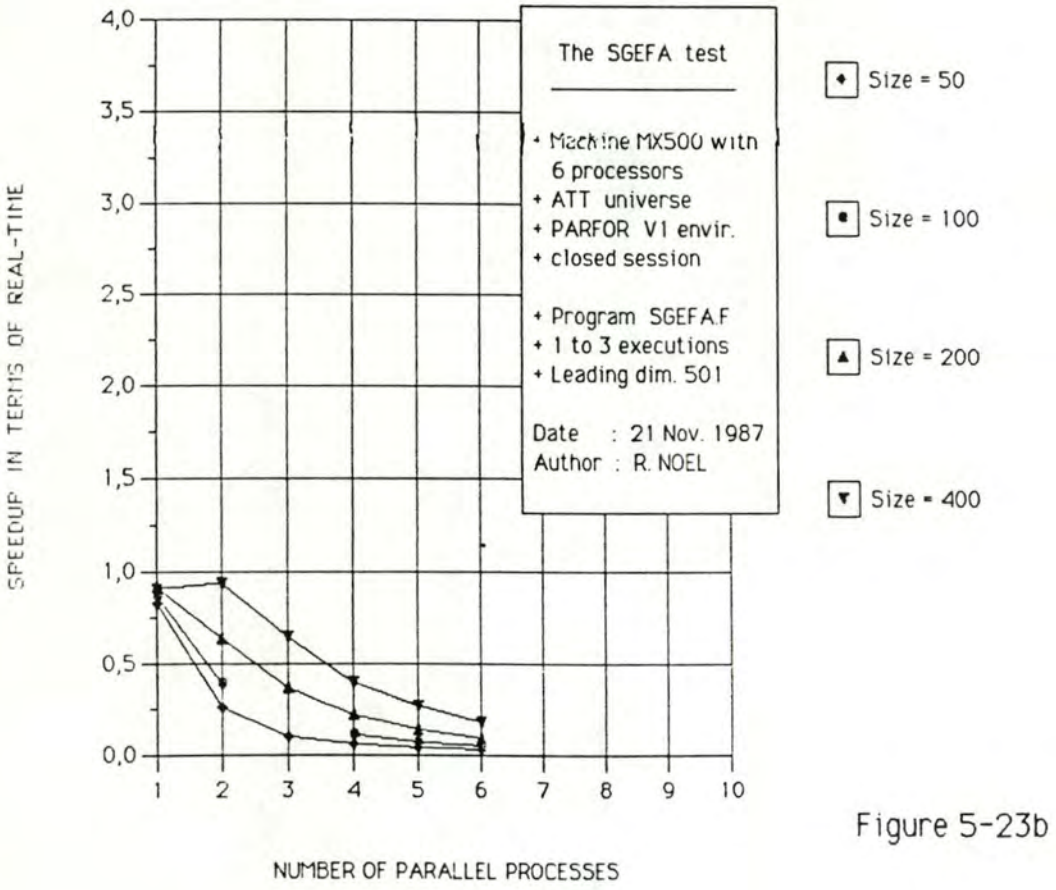


Figure 5-23b

Tests with the version 2 of PARFOR

In this section , with the same program and the same measurements as in the previous section , we can observe a similar behaviour of PARFOR , except that some results are positive when the granularity of the parallel task is increased . For sizes less than 100 , it is never usefull to run a program with more than 1 process .

Anyway , for greater sizes , the speedup is greater than 1 . So , if we increase the size of the parallel job , there is a tendency for the curves to become linear .

Because the test program is the same , we can think that this increase of performances is due to the modified PARFOR environment .

The graphics on figures 5-24 a & b are nearly the same for the Real-Time and the Cpu-time .

Tests with the version 5 of PARFOR

The same program parallelized in the same way , executed in this last version of PARFOR , provides better results . The graphics of figures 5-25a & 5-25b show that the influence of the PARFOR implementation is very high in this kind of parallelization . The costs of the calls to TASKIN are reduced (as we have seen in the serie of tests concerning the overheads) , and this is easely visible on the performance point of vue .

The performances are however still better for large sizes of the parallel subroutines .

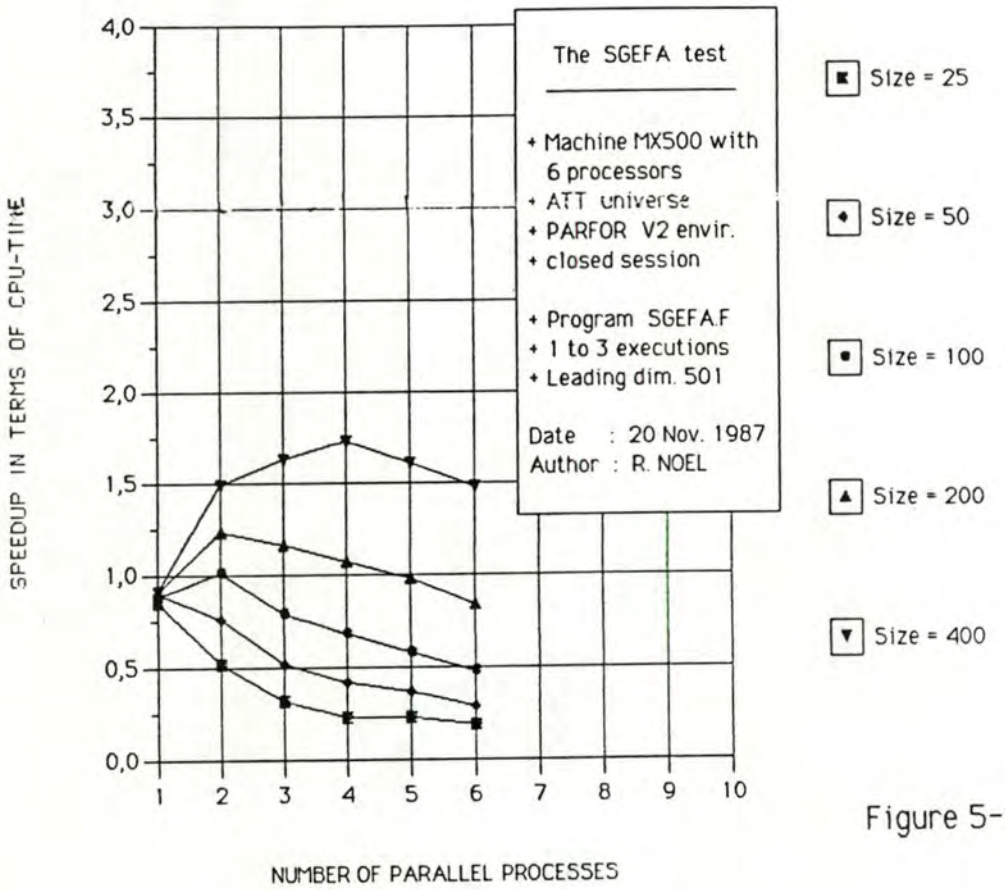


Figure 5-24a

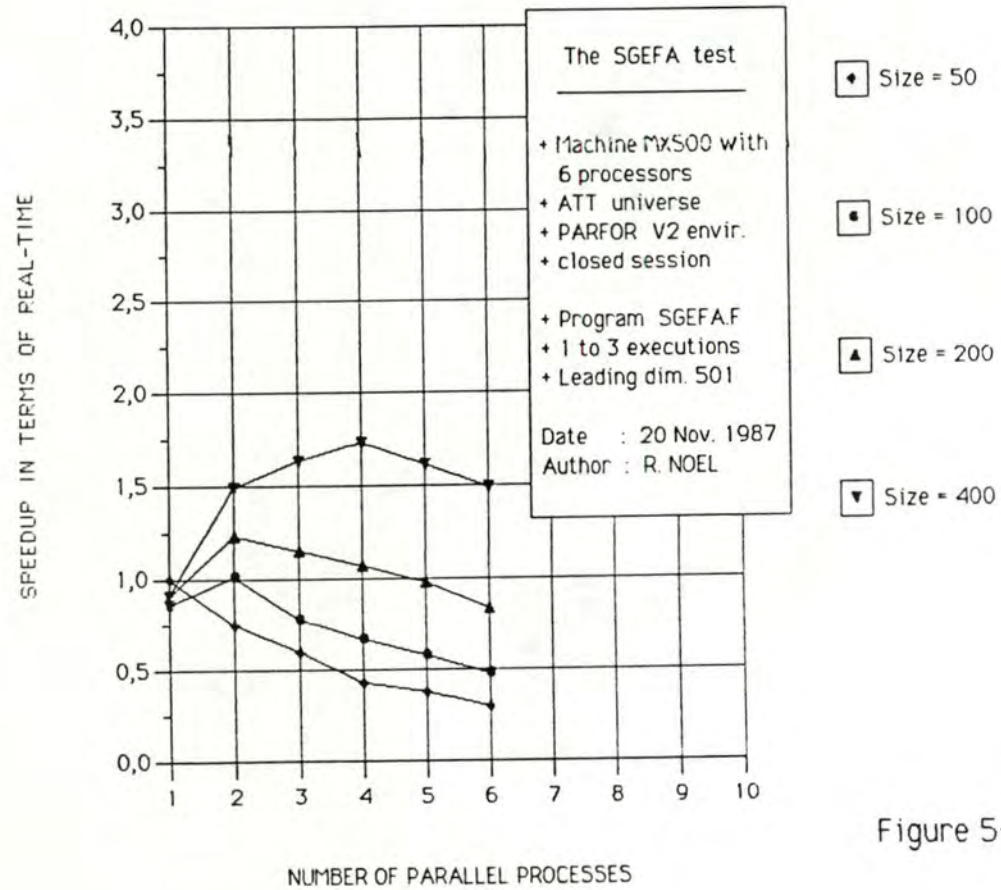


Figure 5-24b

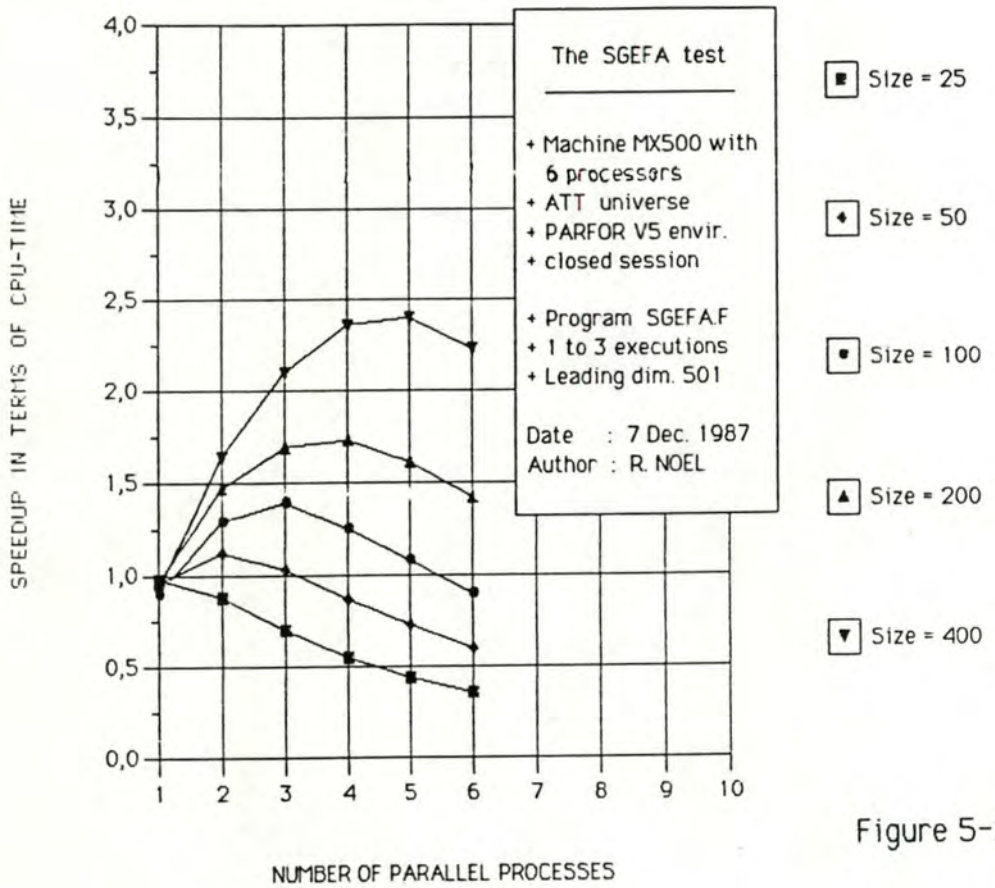


Figure 5-25a

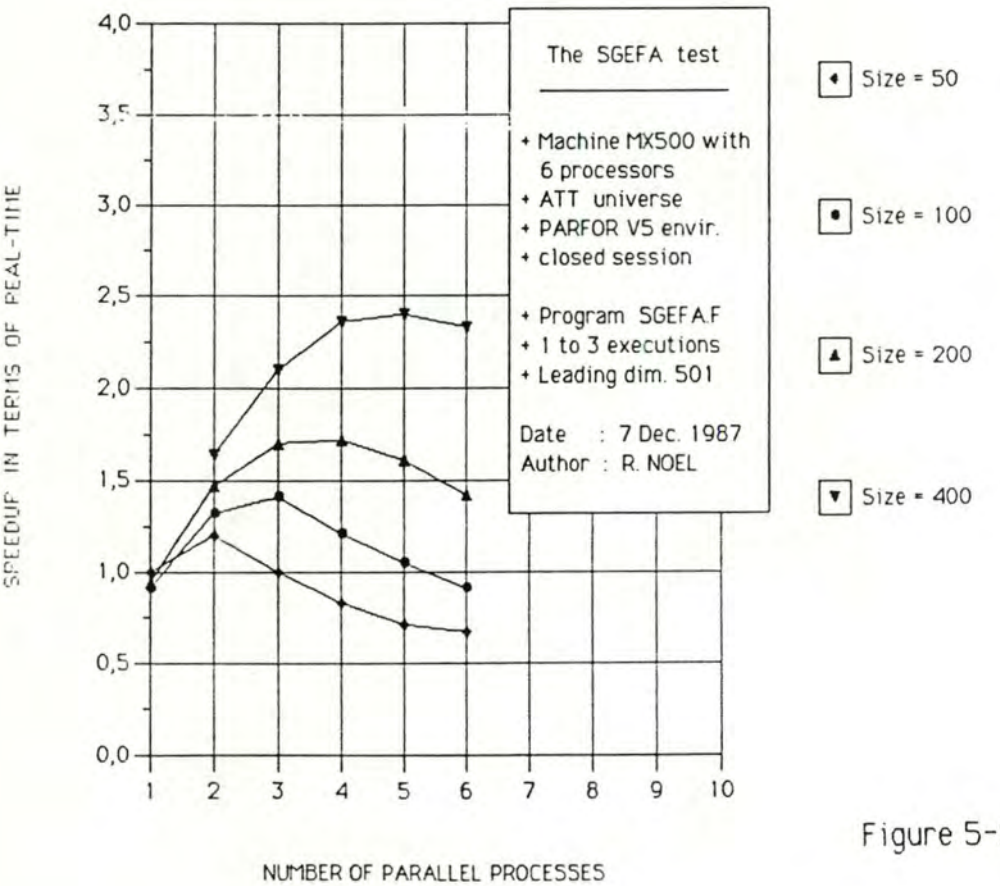


Figure 5-25b

Conclusion for this second level

These results show that this level of parallelization is not very different from the first level . The main difference is located in the number of intermediate calls to subroutines , which is reduced . But this reduction does not affect in a very large way the results , because the extra-calls of the first level , eventually , are not very expensive compared to the more critical time consumption of the TASKIN facility .

This second level of parallelization is very sensible to the costs of the TASIN facility . This is due to the multiple calls that are made to this tool , and that is the reason why we try to reduce the cpu-time of the calls to TASKIN .

The tests we built show that the calls to the TASKIN facility remain very expensive in a parallel program . This is the reason why we try to avoid them as most as possible . This reduction of the calls to the TASKIN facility is done in the third level of parallelization . We discuss the results of ths third method in the next sub-section .

5.10.9 Results of the tests with the third level of parallelization

Tests with the version 1 of PARFOR

The first graphic (figure 5-26a) shows immediately that this third level of parallelization of a PARFOR program is far better for our SGEFA program . The cpu curve shows that the speedup is quasi linear according to the number of processes that are used to run the parallel program .

This curve shows the first good results obtained with the PARFOR environment . For this application , where the size of a parallel job is very little , we can observe that the speedup follows the curve that we expected to see . The speedup is linear until the executions with 4 processes . After that , it is still increasing for most of the curves , but there is a tendency to a certain stabilization .

We can make a distinction between 2 kinds of curves on this graphic .

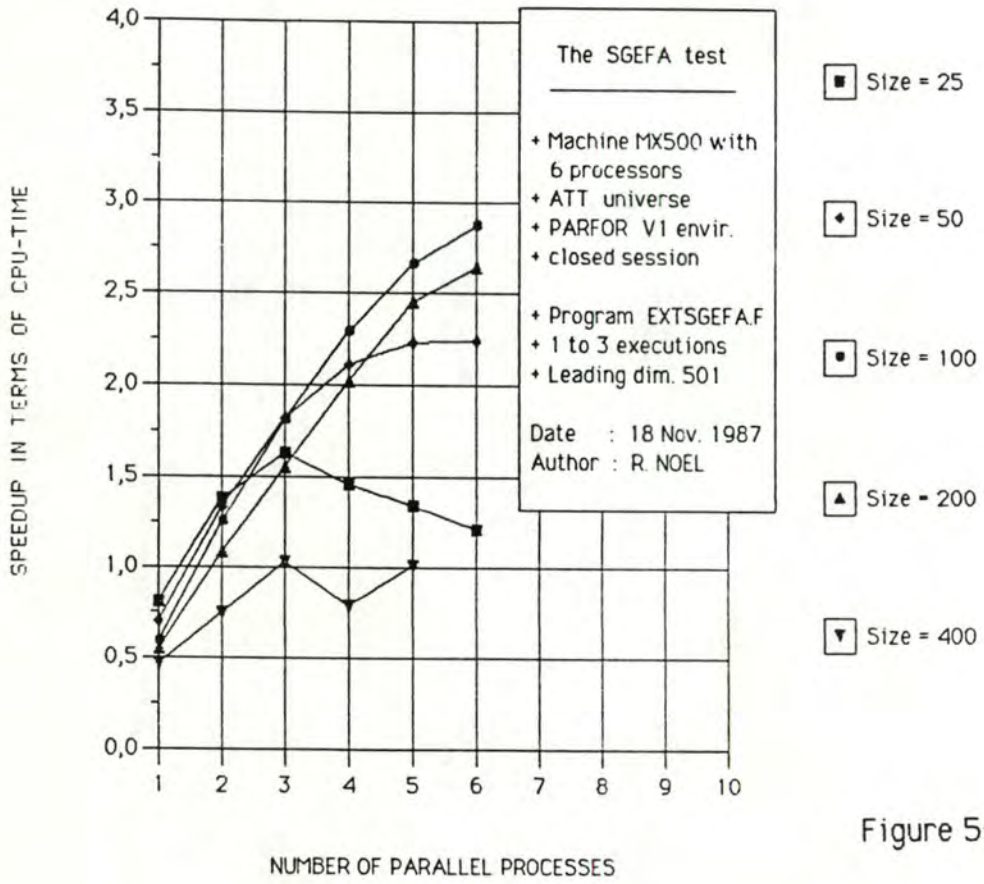


Figure 5-26a

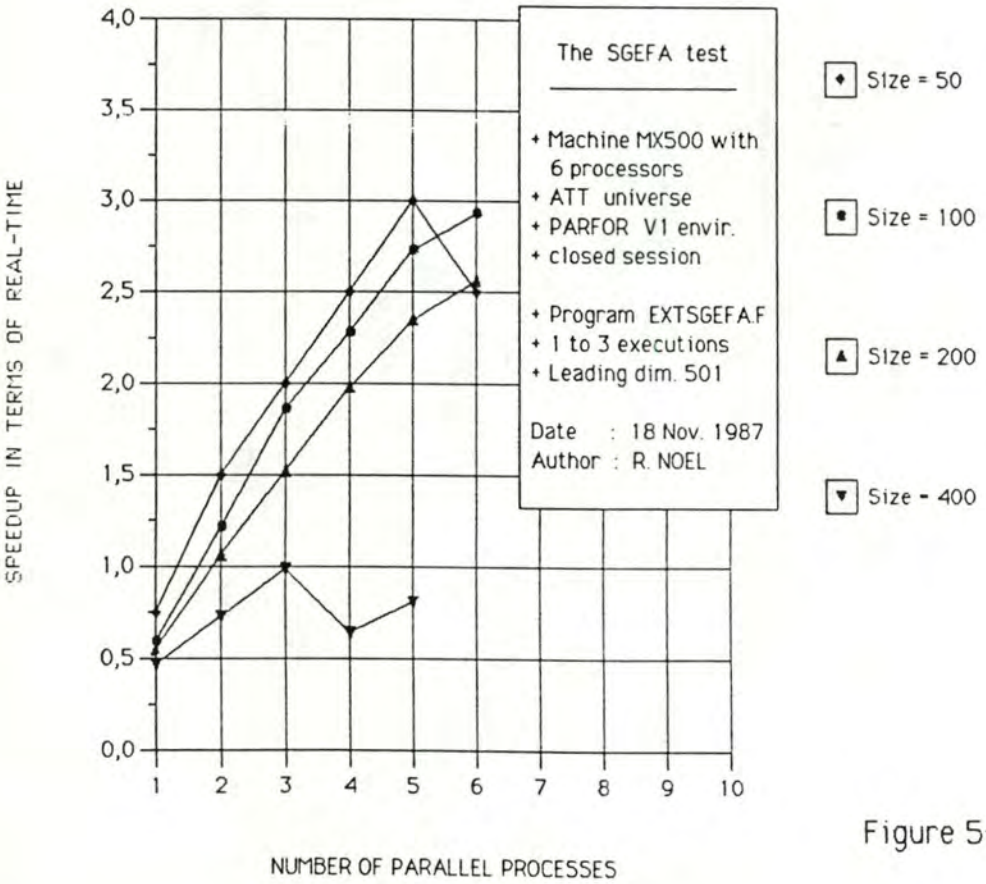


Figure 5-26b

First , the curves drawn for the executions of the algorithm with a matrix size of 50 to 200 . These curves show a relatively good behaviour of PARFOR . The speedup is quasi linear . The stabilization for the executions with more than 4 processes can be explained by the fact that the machine on which the tests are made , disposes of only 6 processors . But in a Unix environment , even if a user is working alone on the system , some processes , part of the system are still running periodically . It implies that , when a parallel program is executed with a number of parallel processes close to the number of available processors , some processors are not immediately available , involving some delays for the user processes , having in turn an influence on the time consumption of the user program .

This is true for the real-time of the parallel program , but in this level of parallelization , it is also true for the cpu-time partially determined by the waiting times at the synchronization barriers . There , the parallel processes are waiting in a busy loop involving unusefull time wasting that can not be avoided . This leads immediately to a direct conclusion that such a parallel program should never be executed with a number of processes that is greater than the available number of processors . That is also the reason why we limited our tests to executions with as a maximum 6 processes , although it was pefectly correct to use more .

Concerning this first serie of curves , we can also notice that the speedup for the execution with 1 process is not 1 . The reason of this is that the comparisons are based on the timings of the sequential algorithm , and not on the timing of the parallel algorithm executed with 1 process . We discussed this subject earlier . Note that taking this assumption into account , it is very clear on the graphic , that the factor of the speedup is approximately equal to the number of processes that execute the program . This means a very good behaviour of PARFOR .

Secondly , the curves drawn for the executions of the algorithm with little matrix sizes , and large matrix sizes . These curves show a relatively bad behaviour of PARFOR for executions with more than 3 processes .

For the little matrix sizes , it seems that the size fo the parallel code is too little compared to the number of operations that are necessary to manage these parallel processes . This implies that most of the time is

spent in the synchronization barriers , wasting such a way the effective user time . This involves a drastic diminution of the performances when the number of parallel processes increases . This phenomenon is relatively similar to the one we discovered in the second type of parallelization with the TASKIN routine . It is the problem of granularity which appears again at a lower level .

For the large matrix size (400) , we have a very strange curve for which we have no reasonable explanation until now . The speedup is not too bad until 3 processes , but then it is stabilized .

It seems very strange that the speedup is so bad , despite the large parallel work assigned to each process . This behaviour goes to the opposite direction with the results we had until now . Perhaps it is a problem of the algorithm , but this would be strange too because the algorithm is independent of such a factor .

One possibility to explain this strange behaviour would be the following : some extra-mechanisms could be automatically called , like paging of the memory , inactivation of the local cache memories , and so on , because of the size of the matrix . But these things are difficult to certify . What could be done to prove it , is to execute the program many times with increasing sizes until the moment at which we can observe large degradations in the performances of the algorithm . But this was not done because of the costs that such tests involve in terms of time .

The curve that is provided for the real-time has nearly the same scheme as the curve provided for the cpu-time . This behaviour can be considered as "normal" because we always use a number of calls to TASKIN which is the same as the number of processes , which in turn , is less or equal to the number of processors. We describe in chapter 3 the influences and the relations between these factors .

Tests with version 2 of PARFOR

In these tests , with the same program and the same measurements as in the previous case , we can observe a similar behaviour of PARFOR . The curves of figures 5-27a & 5-27b show that the performances of this level

of programming are independent of the implementation of the TASKIN facility and the way the messages are sent to the parallel processes .

This behaviour is quite normal , because this way of programming the algorithm implies only the number of times a TASKIN call , that there are parallel processes . The amount of time spent in the TASKIN facility is reduced to a fixed value undependent of the matrix size , and thus , to a minimum . And the modifications made in its implementation do quasi not affect the timing results of the algorithm .

This is why this third method of programming in the PARFOR environment appears to be the best solution . The implication is that PARFOR can also be performant for small granularity parallel programs , if they are parallelized in such a way , with barriers when the synchronizations are necessary , and with shared memory as the mean of communications between the processes .

Tests with the versions 4 and 5 of PARFOR

The same remarks from the PARFOR environment 3 are still valuable for the results provided by the same program executed with the versions 4 and 5 of the PARFOR . The differences in the implementation of PARFOR do affect in a very little and constant way , the results of the algorithm under test . This affection is negligible compared to the execution time of the algorithm .

Some variations can be observed on the curves (figures 5-28 & 5-29) . But these are very little , and can be considered as acceptable statistic variations .

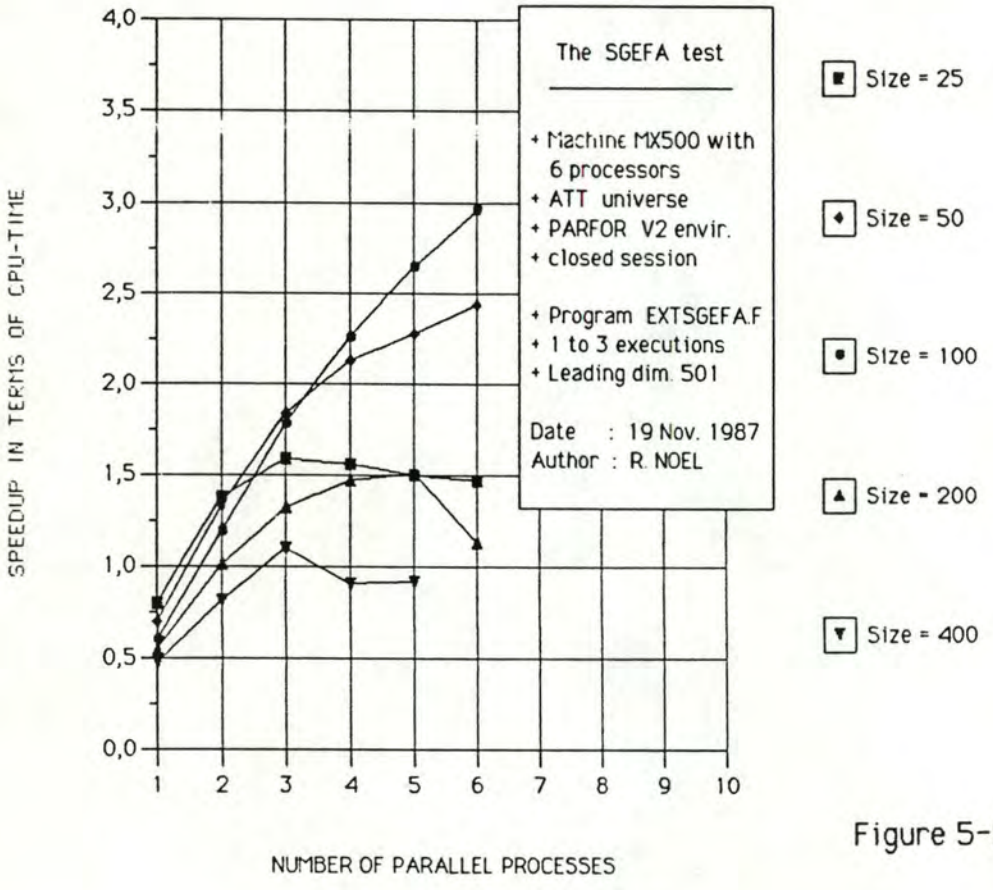


Figure 5-27a

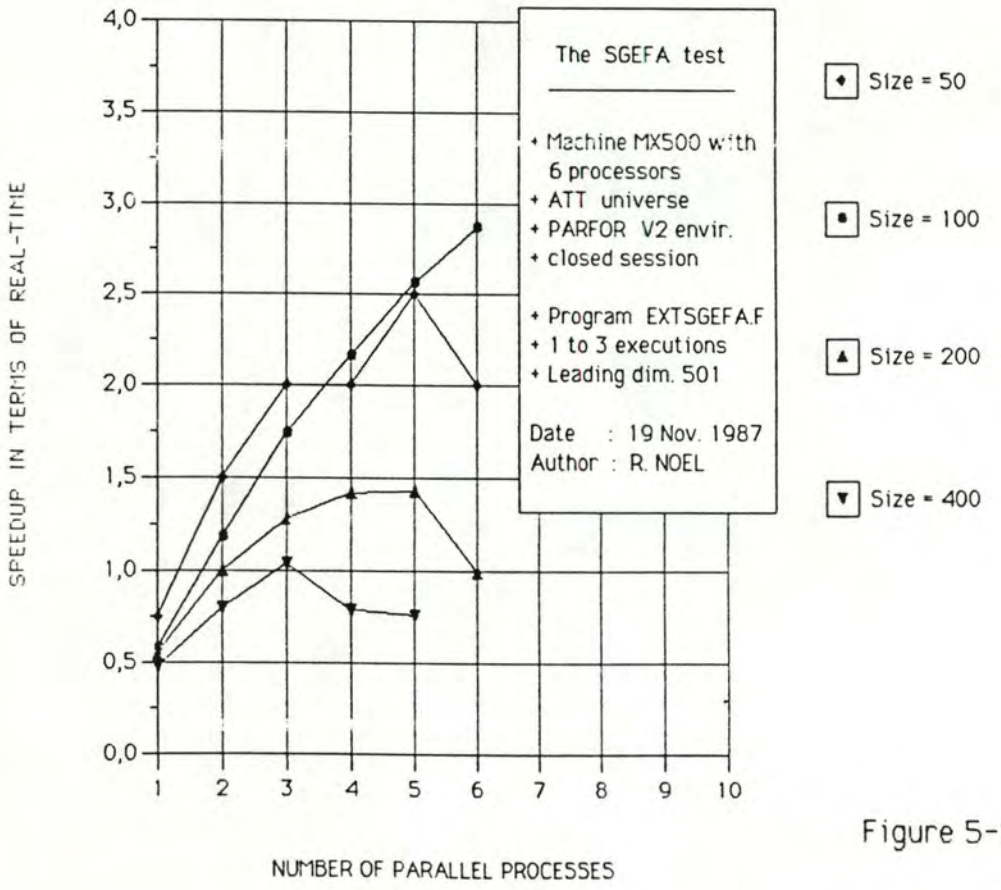


Figure 5-27b

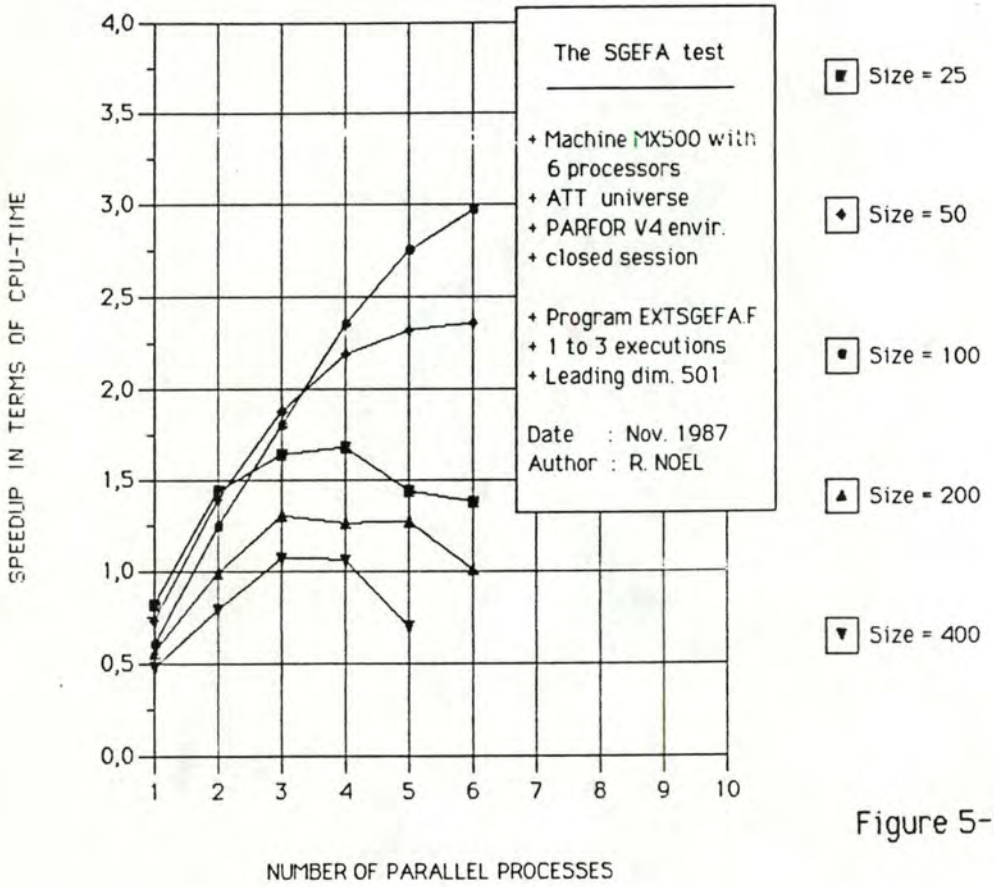


Figure 5-28a

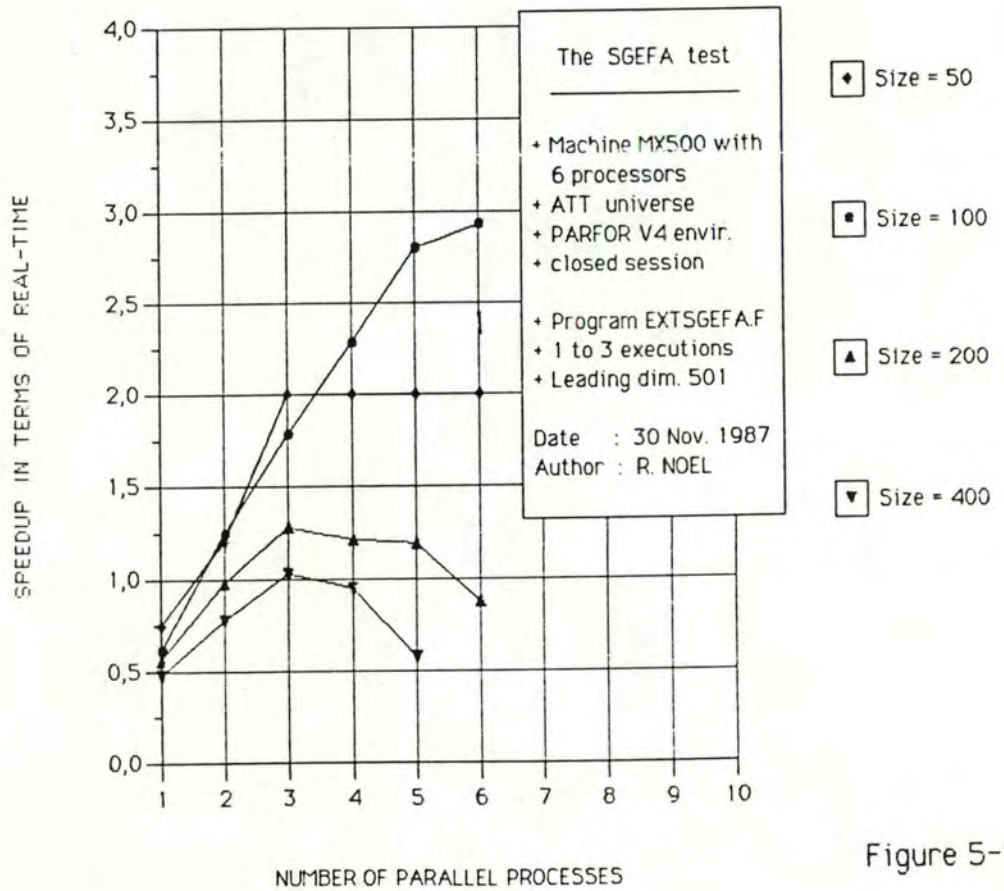


Figure 5-28b

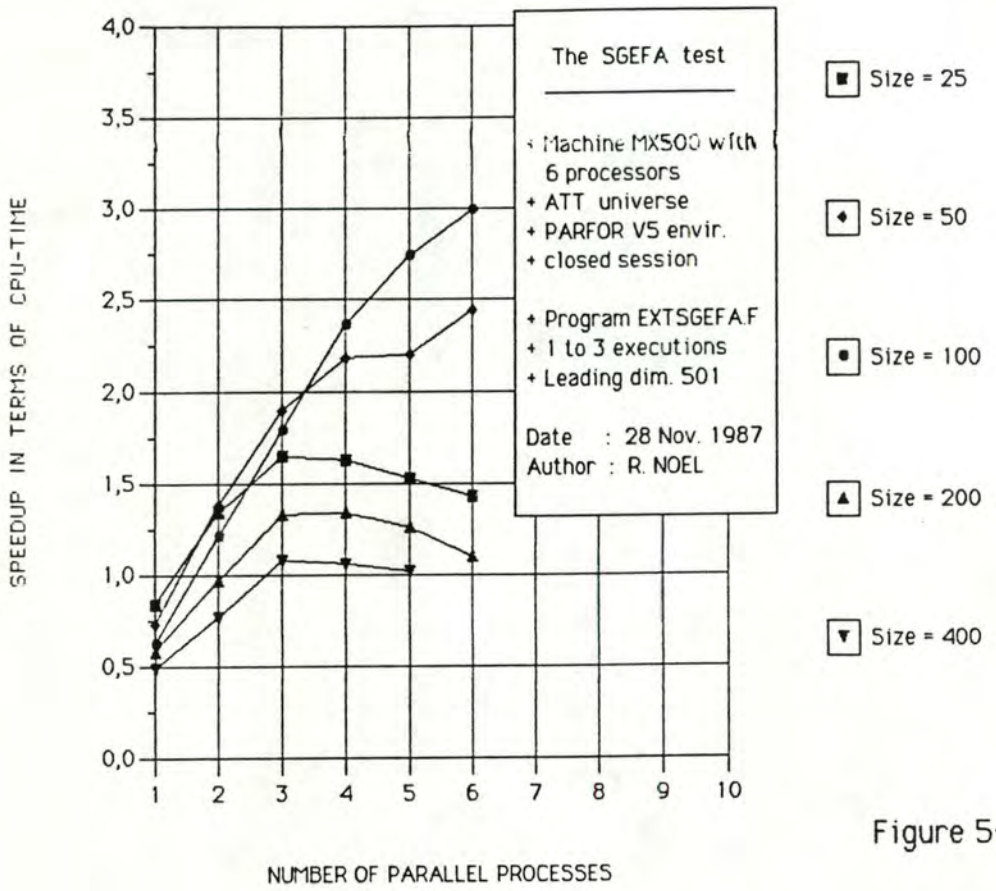


Figure 5-29a

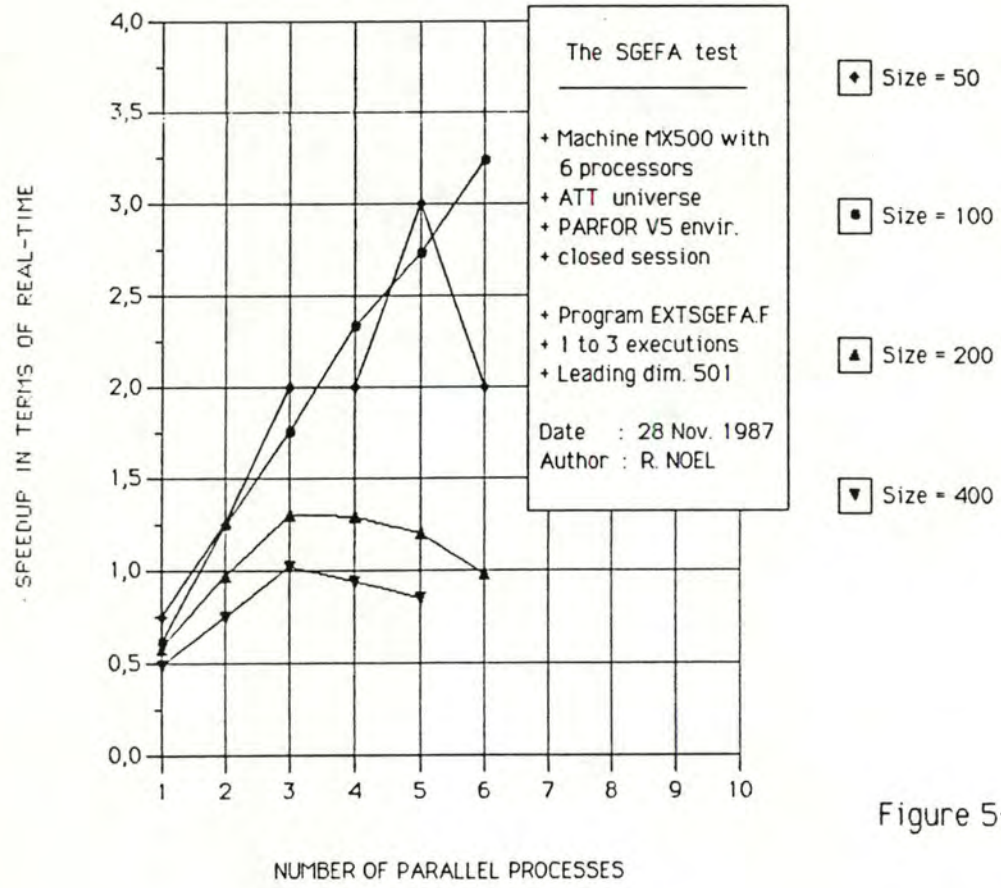


Figure 5-29b

Tests with the FORCE environment

The FORCE environment provides tools that are relatively different from those provided by PARFOR . FORCE is actually considered as " the state of the art " for parallel programming in FORTRAN like languages , while PARFOR is just born some months ago .

The FORCE SGEFA version that we tested on our machine was written and provided to us by professor H. Jordan . We did not modify anything in this parallel algorithm . We have only modified the enclosing program in such a way that the timing results are provided in the same standard tables we proposed for the PARFOR environment .

The results provided on the graphics show very good performances for the SGEFA routine. In fact , because of the philosophy of the FORCE environment , a FORCE program has necessary the same structure that we described for the third method of parallelization within PARFOR . For this reason , even if FORCE provides many tools for parallel programming , the performances of a program can not be very different from the performances provided by PARFOR . The main differences that remain between the SGEFA program written in FORCE and the SGEFA program written in PARFOR are explained in the section describing the third level of parallelization . Mostly , FORCE makes extensive uses to hardware facilities . But this is implicit for the FORCE user , which only write macros for the preprocessor .

Then , the results are no more dependent of the environment , but well of the use or not of the hardware facilities . If they are used , because of their specificity to the machine , they must increase the performances of a program that makes use of them against the same program that do not use these functions .

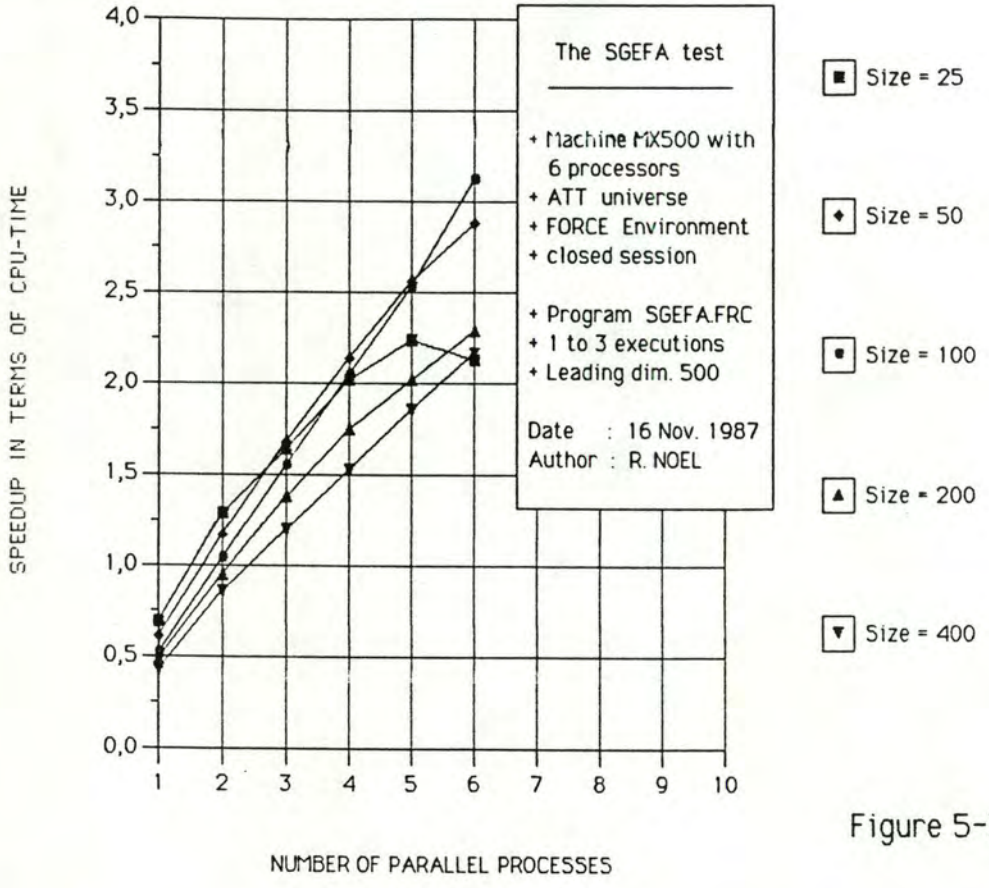


Figure 5-30a

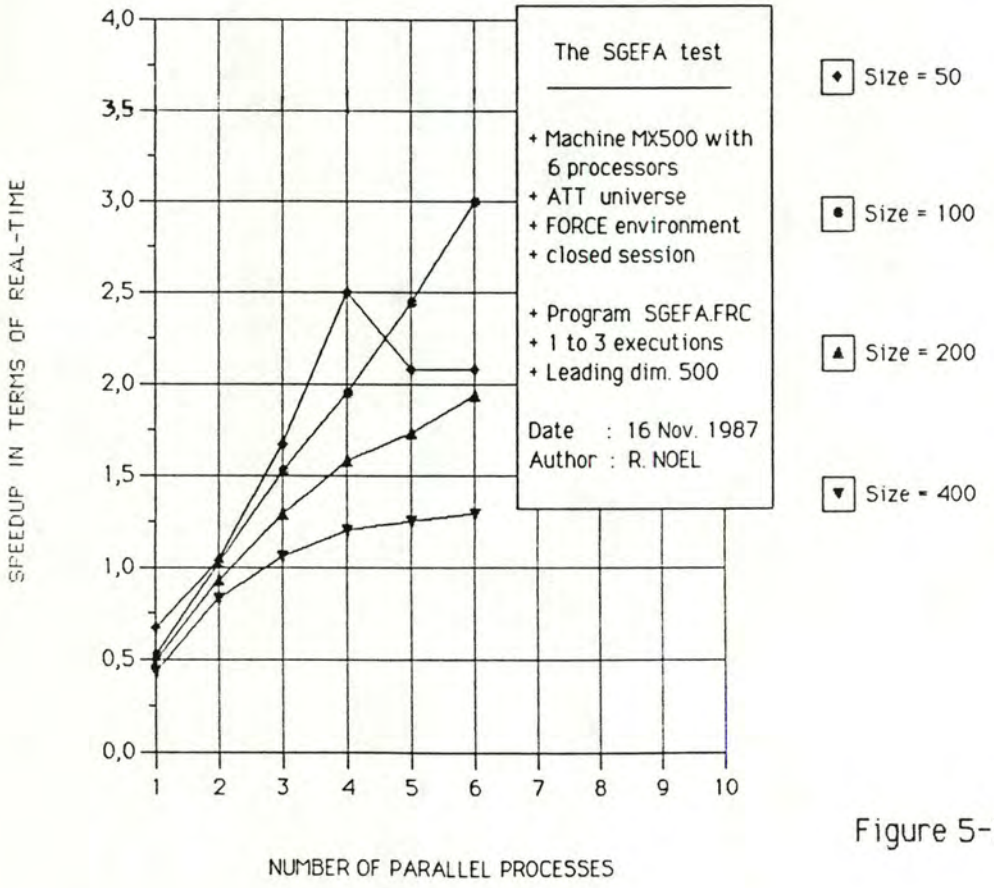


Figure 5-30b

Conclusion for this third level

As we have seen , this third level of parallel programming seems to be the best way to program in PARFOR . The curves show that the speedup is nearly always good , despite of some exceptions that we can explain or not. This way of programming requires additionnal tools that can be directly implemented in FORTRAN , perticulary the barriers that allow an easy way for synchronizations . These barriers seem to be sufficient for many parallel programs , but are expensive in terms of time consumption . The curves for the same FORCE program are better than the curves for PARFOR . But this is mainly due to the fact that FORCE uses hrdware atomic locks , while PARFOR not .

The main characteristic of this way of programming is that we can really say that it is parallel programming . The most important things in a parallel program are not the calls to the TASKIN routine , but well the synchronizations between the processes .

Chapter 6

Some final words as conclusion

6.1 Introduction

In this work , we studied concepts , both for the starting point of the development of the PARFOR environment , and for programming within this environment . We also made some comparisons with the FORCE environment - that we had first to adapt to our UNIX system - and its concepts .

At the beginning of this study , after the first tests , we were disappointed at the view of the results . They were rather bad . But further tests more elaborated allowed us to reconsider the implementation on some specific points , to increase the speed of the parallelized applications , and eventually , get better results .

The tests we made were essentially concentrated on benchmark programs , and this could be considered as a restriction to the results that we got . However , the LINPACK application which was the base of our work , is a complete scientific application , but still compact , compared to some others . An extention of this work could be the parallelization of larger applications to confirm the results we have , for very large parallel programs .

6.2 A future for PARFOR ?

The main hope concerning this study , is that the time spent to this project is not spent "just for building one another of these multiple tries in FORTRAN-like parallel computing languages" . We can already confirm that , this project is continuing in collaboration with the TU München , and another student , trying to adapt PARFOR in another context (VAX clusters environment) . The PARFOR environment , is also developped for the BS2000 machines in another team of our department . The key words for the futur of PARFOR remain SIMPLICITY and PERFORMANCES .

Simplicity for the user to have the possibility to modify its applications with a low number of tools .

Performances for the applications to gain in time when executed within this parallel system .

6.3 My personal experiment

Anyway , I found very interesting to work in this area of parallel programming , and interesting to discover practical difficulties that parallel programs involve . But in all cases , I found and realized that the parallel programming is always more complicated than the traditionnal sequential programming , leading to a greater time in trying to find a parallel solution to a given algorithm , and to a longer time in debugging a program that includes parallelism .

Bibliography

[Abstreiter] F Abstreiter
PARFOR : Paralleles FORTRAN : Entwurf des prototyps
Institut für Informatik , TU München , 1987

[Abstreiter] F. Abstreiter
PARFOR - Paralleles FORTRAN : Implementierung unter BS2000 .
Institut für Informatik TU München , internes Papier , August 1987

[Anderson-Jensen] George A. Andersen & E. Douglas Jensen
Computer interconnection structures : Taxonomy , characteristics and examples .
Computing Surveys , vol. 7 , n°.4 , dec 1975 , p 197 - 213

[Ardoni-Boccalatte-DiManzo] G. Adorni , A Boccalatte & M Di Manzo
Evaluation of scheduling algorithms in the multiprocessor environment
Computer Performance volume 2 (1981) p 70 - 76

[Axelrod] Tim S. Axelrod
Effects of synchronization barriers on multiprocessors performances
Parallel Computing volume 3 (1986) p 129 - 140

[Bach] Maurice J. Bach
The design of the UNIX operating system
Prentice-Hall , 1986

[Ben Ari] Ben Ari M.
Principles of concurrent programming
1982

[Benwell] Nicholas Benwell , 1975
Benchmarking : Computer evaluation and measurement
John Wiley & Sons , 1975

[Brice] R. Brice
Benchmarking your benchmarks : A user 's perspective
Computer Performance volume 4 (1983) p 73 - 79

[Browne] J. C. Browne
Framework for formulation and analysis of parallel computation structures
Parallel Computing volume 3 (1986) p 1 - 9

[Brumfield] J. A. Brumfield
Operationnal response time formulas and their sensitivity to errors
Parallel Computing volume 3 (1986) p 93 - 110

[Buzbee] B. L. Buzbee
A stategy for vectorization
Parallel computing volume 3 (1986) p 187 - 192

[Bytheway] A. J. Bytheway
On the proper treatment of measurement data
Performance volume 1 (1980) p 28 - 36

[Calahan] D. A. Calahan
Task granularity studies on many-processor CRAY X-MP
Parallel Computing volume 2 (1985) p 109 - 118

[Carnevali-Sguazzero-Zecca] P. Carnevali, P. Sguazzero, V. Zecca
Microtasking on IBM multiprocessors
IBM J. Res. Develop. Vol. 30, N°6 , november 1986

[Cavouras] J. C. Cavouras
Simulation of user processes
Computer Performance volume 2 (1981) p 192 - 195

[Chu-George] Eleanor Chu & Alan George
Gaussian eleimination with partial pivoting and load balancing on a multiprocessor .
Parallel Computing volume 5 (1987) p 65 - 74

[Clausing-Hagstrom] J.A. Clausing , R. Hadstrom, E.L. Lusk & R.A. Overbeek
A technique for achieving portability among multiprocessors : Implementation on the Lemur .
Parallel Computing volume 2 (1985) p 137 - 162

[Darema-Rogers] F. Darema - Rogers
Parallel applications development and performance
IBM Europe Institute , parallel computing , August 1986

[Denning] Peter J. Denning
Parallel computing and its evolution
Communications of the ACM, vol. 29, n°.12, dec 86, p 1163-1167

[Dongarra-87] Jack J. Dongarra & Lennart Johnsson
Solving banded systems on a parallel processor
Parallel Computing volume 5 (1987) p 219 - 246

[Dongarra-Sameh-Sorensen] J.J. Dongarra , A.H. Sameh , D.C. Sorensen
Implementation of some concurrent algorithms for matrix factorization
Parallel Computing volume 3 (1986) p 25 - 34

[E.Allen] Frances E. Allen
Compiling for parallelism
IBM Europe Institute , parallel computing , August 1986

[Eichholz] Stefan Eichholz
Parallel programming with PARMOD
Institut für Informatik TU München , August 1987

[Elisabeth-Hull-Donnen] M. Elisabeth , C. Hull and G. Donnen
Contextually communicating sequential processes : A software engeneering environment
Software : Practice and Experience volume 16 (1986) p 845 - 864

[Faber-Lubeck-White] V. Faber , Olaf M. Lubeck , Andrew B. White , Jr
Superlinear speedup of an efficient sequential algorithm is not possible
Parallel Computing volume 3 (1986) p 259 - 260

[Faber-Lubeck-White-87] V. Faber , Olaf M. Lubeck , Andrew B. White
Comments on the paper " Parallel efficiency can be greater than unity .
Parallel Computing volume 4 (1987) p 209 - 210

[Ferrari-Spadani] D. Ferrari & M. Spadani
Experimental computer performance evaluation

[Fisher] A. J. Fischer
A multiprocessor implementation of occan
Software : Practice and Experience volume 16 (1986) p 875 - 892

[Frederickson-Jones-Smith] P. O. Frederickson , R. E. Jones & B. T. Smith
Synchronization and control of parallel algorithms
Parallel Computing volume 2 (1985) p 255 - 264

[Fuller] H. Fuller, K.Ousterhout
Multi-microprocessors: An overview and working examples
Proceedings of the IEEE , vol. 66 , n°.2 , february 1978, p 216- 228

[Gabriel] R. P. Gabriel , MIT press , 1985
Performance and evaluation of LISP systems
Prentice Hall , 1986

[Gait] J. Gait
A probe effect in concurrent programs
Software : Practice and Experience volume 16 (1986) p 225 - 233

[Gajski-Peir] Daniel D. Gajsky & Jih-Kwon Peir
Comparison of Five multiprocessor systems
Parallel Computing volume 2 (1985) p 265 - 282

[Gehani-Roome] N. H. Gehani and W. D. Roome
Concurrent C
Software : Practice and Experience volume 16 (1986) P 821 - 844

[Ghezzi] C. Ghezzi
Concurrency in programming languages : A survey
Parallel Computing volume 2 (1985) p 229 - 241

[G-Schmitt] Günter Schmitt
Fortran-Kurz technisch orientiert : Einführung in die programmierung mit Fortran 77
München Wien , 1985

[Haring] G. Haring
Workload characterization at task level
Computer Performance vomume 3 (1982) p 61 - 72

[Hillis-Steele] W. Daniel Hilis & Guy L. Steele
Data parallel algorithms
Communications of the ACM, vol. 29, n°.12, dec 86, p 1170-1183

[H.F. Jordan] H. F. Jordan
Structuring parallel algorithms in an MIMD shared-memory environment
Parallel Computing volume 3 (1986) p 93 - 110

[Hobfeld-Weider] F. Hosfeld & P. Weider
Parallele algorithmen
Informatik-Spektrum (1983) , vol. 6 , p 142 - 154

[Hockney-84] R. W. Hockney
MIMD computing in the USA 1984
Parallel Computing volume 2 (1985) p 119 - 136

[Hockney-86] R.W. Hockney
Parametrization of computer performance
IBM Europe Institute , parallel computing , August 1986

[Hornstein] J. Virgil Hornstein
Parallel processing attacks real-time world
Mini-Micro systems , december 86 , p 65

[Horton-Turner] I. A. Horton and S. J. Turner
Using coroutines in Pascal
Software : Practice and Experience volume 16 (1986) p 45 - 61

[Hossfeld] F. Hossfeld
Strategies for parallelism in algorithms
IBM Europe Institute , parallel computing , August 1986

[Janssen] R. Janssen
A note on superlinear speedup
Parallel Computing volume 4 (1987) p 211 - 213

[Janssens-Annot-Goor] M.D. Janssens, J.K. Annot, A.J. Van de Goor
Adapting Unix for a multiprocessor environment
Communications of the ACM, vol. 29, number 9, sep 86, p 895-901

[Johnstone] Ian Johnstone
Strength in numbers
Unix Review , february 1986 , p 53 - 58

[Jones-Schwarz] Anita K. Jones & Peter Schwarz
Experience using multiprocessor systems - a status report
Computing Surveys , vol. 12 , n°.2 , june 1980 , p 121 - 213

[Jordan-87-A] Harry F. Jordan
FORCE user's manual
Department of electrical and computer engeneering , uneversity
of Colorado , 1987

[Jordan-87-B] Harry F. Jordan
The FORCE
Department of electrical and computer engeneering , uneversity
of Colorado , 1987

[Jordan-87-C] Harry F. Jordan , Norbert S. Arenstorf
Comparing barrier algorithms
Department of electrical and computer engeneering , uneversity of Colorado , 1987

[Jordan-87-D] Harry F. Jordan
Interpreting parallel processor performance measurements
Department of electrical and computer engeneering , uneversity of Colorado , November 1985

[Kerridge-Simpson] Jon Kerridge & Dan Simpson
Communicating parallel processes
Software : Practice and Experience volume 16 (1986) p 63 - 86

[Knight] A. J. Knight
Can measurement ever be justified ?
Performance volume 1 (1980) p 120 - 124

[Kogge] P. M. Kogge
Function-based computing and parallelism : A review
Parallel Computing volume 2 (1985) p 243 - 253

[Kutti] S. Kutti
Taxonomy of parallel processing and definitions
Parallel Computing volume 2 (1985) p 353 - 359

[Manna-Pnueli] Z. Manna & A. Pnueli
Verification of concurrent programs : the temporal framework
International Lecture Series in Computer Science , 198? , p 215-273

[Morris-Roth] M. F. Morris , P. F. Roth
Tools and techniques : Computer performance evaluation for effective analysis
1982

[Ohbuchi] R. Ohbuchi
Overview of parallel processing research in Japan
Parallel Computing volume 2 (1985) p 219 - 228

[Parkinson-86] D. Parkinson
Parallel efficiency can be greater than unity
Parallel Computing volume 3 (1986) p 261 - 262

[Patrick-Reed-Voigt] Merrell L. Patrick, Daniel A. Reed, Robert G. Voigt
The impact of domain partitioning on the performance of a shared memory multiprocessor .
Parallel Computing volume 5 (1987) p 211 - 217

[Pedersen] T. Pedersen
Process administration at a high level language
Software : Practice and Experience volume 16 (1986) p 303 - 333

[Reddi] A. V. Reddi
Performance of pipeline and parallel architectures for communication processors
Computer Performance volume 5 (1984) p 102 - 107

[Reed-Patrick] D. A. Reed & M. L. Patrick
Parallel iterative solution of sparse linear systems : Models and architectures
Parallel Computing volume 2 (1985) p 45 - 67

[Reiter] E. Reiter
Some new ideas for parallel algorithms on vector and multiprocessors
California State University , Hagward , dec 1986

[Rettberg-Thomas] Randall Rettberg & Robert Thomas
Contention is no obstacle to shared memory multiprocessing
Communications of the ACM, vol. 29, number 12, dec 86, p 1202-1212

[Sabatier] A. Sabatier
Le multibus et ses signaux
Micro-Informatique , février 1979 , p 3 - 11

[Sequent] Sequent company
The balance 8000 parallel computer : Guide to parallel programming

[Siemens-C] Siemens AG
Programmiersprache für Fortgescrittene
Siemens München , version 3.0 , Januar 1987

[Sonnenschein] M. Sonnenschein
An extention of the language C for concurrent programming
Parallel Computing volume 3 (1986) p 59 -71

[SPAB_T] SPAB_T
SW und HW Architektur der MX500
D ST SP 5 , Siemens AG , internes Dokument , August 1987

[Tandem] Tandem company
Les systèmes fault tolerant
Conférence donnée à Namur , avril 1987

[Terplan] K . Terplan
Real time performance management
Performance volume 1 (1980) p 16 - 21

[Van Lamsweerde] A. Van Lamsweerde
Cours sur les systèmes répartis
FNDP Namur , Année accadémique 1986-1987

[Wood] A. M. Wood
The organization of parallel processing machines
Wopplot 83 : Parallel Processing 1983

[Yuba-Kashiwagi] Toshitsugu Yuba & Hiroshi Kashiwaga
The Japanese national project for new generation supercomputing systems
Parallel Computing volume 4 (1987) p 2 - 16